Michael Simões Silva

Bachelor in Computer Science

Validation of Data Invariants in the OutSystems Platform

Dissertation submitted in partial fulfillment
of the requirements for the degree of

Master of Science in
Computer Science and Engineering

Adviser: João Ricardo Viegas da Costa Seco,
Assistant Professor, NOVA University of Lisbon

Co-adviser: Hugo Miguel Ramos Lourenço,
Software Engineer, OutSystems

Examination Committee

Chair: Dr. António Ravara, Associate Professor, NOVA University of Lisbon
Members: Dra. Maria Antónia Lopes, Associate Professor, University of Lisbon
Dr. João da Costa Seco, Assistant Professor, NOVA University of Lisbon

FACULDADE DE
CIÊNCIAS E TECNOLOGIA
UNIVERSIDADE NOVA DE LISBOA

December, 2019

Validation of Data Invariants in the OutSystems Platform

This document was created using the (pdf)LaTeX processor, based on the "novathesis" template[1], developed at the Dep. Informática of FCT-NOVA [2].
[1] https://github.com/joaomlourenco/novathesis [2] http://www.di.fct.unl.pt

Acknowledgements

I would like to thank Faculdade de Ciências e Tecnologias from the Universidade Nova de Lisboa for giving me the fundamental tools over the past five years that will now allow me to start my professional career. Thank you also to OutSystems for funding a scholarship for this dissertation.

Thank you to my advisers: João Costa Seco and Hugo Lourenço. I am positive that the outcome of this dissertation would have been very different without your mentoring and help. A special thank you to Hugo Lourenço for closely following my progress and helping with whatever concern I had.

I would also like to thank the other thesis students and colleagues at OutSystems, which I'm glad to call them friends, António Ferreira, David Mendes, Francisco Cunha, Francisco Magalhães, Joana Tavares, João Gonçalves, Lara Borisoglebski, Mariana Cabeda, Miguel Loureiro, Miguel Madeira, Nuno Calejo and Nuno Pulido. During an intensive work year, our discussions, confessions and laughs were essential for helping me have fun and relieve stress.

A special thank you to all my close friends, whom I do not need to mention. Thank you for your love and support, and for being always there for me when things would get a bit depressing. I cannot thank you enough, and I hope our friendships keep only growing stronger as time drags by.

Finally, I want to thank my parents: José Almeida Silva and Maria de Fátima Silva. Without them, I could not be where I am now. Thank you for all the incredible sacrifices you made to provide me with the choice to be who I wanted. I will forever be grateful to you. Last but not least, I would like to thank my sister, Diana, for always being there for me, despite our annoyances and discussions.

Abstract

Software systems collect information from multiple sources, both internal and external. Therefore, it is indispensable that the data should be validated according to data invariants, in all untrusted frontiers of a system. Developers need to define the validation logic for each of those constraints and to write the actual validation code. To safeguard data integrity throughout the entire system, the developers also need to ensure validation in different layers. This obligation can easily head to a problem since it usually leads to duplication of the validation code and contributes to more complex and less structured software architecture, which consequently leads to systems harder to maintain.

The OutSystems platform enables visual development of enterprise Web and Mobile applications, providing an abstraction layer that allows developers to handle the inherent complexity of application development more easily. Still, developers need to write the code responsible for validating data explicitly.

We propose an invariant propagation mechanism capable of propagating data constraints across the various layers of a system, that materialises into the automatic generation of validation code that properly ensures the data integrity in the entire system. Given a set of data constraints, or invariants, defined in the data layer that constrain entities' attributes, our mechanism propagates and manipulates their specification. The propagation is done using a *path-sensitive* data flow analysis technique, specified in Datalog and that uses a Prolog engine as the resolution engine. This solution ensures that the developer only needs to define the data invariants and their logic once in the entire application, and also ensures that it will generate validations whenever necessary. Thus, we remove the programmer's obligation of writing and maintaining validation code, plus ensuring data integrity and providing faster error feedback to users. We evaluated our results in a prototype of the OutSystems platform.

Keywords: Data Validation, Invariants, Data Integrity, Data Flow Analysis, *Path-Sensitive*, Low-code Platforms, OutSystems

Resumo

Os sistemas de *software* colecionam informação de várias fontes, quer internas e externas. Portanto, é indispensável que os dados sejam validados de acordo com invariantes de dados, em todas as fronteiras não confiáveis de um sistema. Os programadores precisam de definir a lógica de validação para cada uma dessas invariantes, e de escrever o respectivo código. Para garantir a integridade dos dados em todo o sistema, os programadores também precisam de garantir a validação em diferentes camadas. Esta obrigação pode facilmente resultar num problema, pois geralmente leva à duplicação do código de validação e contribui para uma arquitetura de *software* mais complexa e menos estruturada, o que consequentemente, resulta em sistemas mais difíceis de manter.

A plataforma OutSystems permite o desenvolvimento visual de aplicações Web e Mobile, fornecendo uma camada de abstração que permite aos programadores lidar com a complexidade inerente ao desenvolvimento de aplicações mais facilmente. Porém, os programadores necessitam de escrever o código responsável pela validação dos dados.

Nós propomos um mecanismo de propagação de invariantes capaz de propagar restrições de dados pelas várias camadas de um sistema, e que se materializa na geração automática de código de validação que garante a integridade dos dados em todo o sistema. Dado um conjunto de restrições de dados, ou invariantes, definidos na camada de dados e que restringem os atributos das entidades, o nosso mecanismo propagará e manipulará as suas especificações. A propagação é feita com recurso a uma técnica de análise estática de fluxo de dados sensível a caminhos de execução, especificada em Datalog e que utiliza um *engine* Prolog como motor de resolução. Esta solução garante que o programador precisa apenas de definir as invariantes dos dados e suas lógicas, uma única vez, garantindo que é gerado validações sempre que necessário. Assim, removemos a obrigação do programador de escrever e de manter esse código de validação, para além de garantiros a integridade dos dados e fornecermos *feedback* mais rápido aos utilizadores. Avaliámos os nossos resultados num protótipo na plataforma OutSystems.

Palavras-chave: Validação de Dados, Invariantes, Integridade de Dados, *Data Flow Analysis*, *Path-Sensitive*, Plataformas *Low-code*, OutSystems

Contents

List of Figures xv

List of Tables xvii

Listings xix

1 Introduction 1
- 1.1 Context and Description . . . 1
- 1.2 Motivation . . . 3
- 1.3 Objectives . . . 4
- 1.4 Key Contributions . . . 5
- 1.5 Running example . . . 5
- 1.6 Document Structure . . . 6

2 OutSystems Platform 7
- 2.1 Architecture . . . 7
  - 2.1.1 Service Studio . . . 7
  - 2.1.2 Integration Studio . . . 7
  - 2.1.3 Platform Server . . . 8
- 2.2 OutSystems Language . . . 8
  - 2.2.1 Screens . . . 9
  - 2.2.2 Actions . . . 9
  - 2.2.3 Entities / Database Model . . . 11
  - 2.2.4 Validations . . . 11

3 Background 15
- 3.1 Defensive Programming . . . 15
- 3.2 Design By Contract . . . 15
  - 3.2.1 Class Invariants . . . 16
- 3.3 Relational Databases - Integrity restrictions . . . 17
  - 3.3.1 Domain Constraints . . . 18
  - 3.3.2 Column Constraints . . . 18
  - 3.3.3 Schema-Level Constraints - *Assertions* . . . 20

3.3.4 User-Defined Constraints - *Triggers* . . . 20
3.4 Type Systems and Refinement Types . . . 21
3.5 Data Flow Analysis . . . 22
3.5.1 Path-Sensitive Data Flow Analysis . . . 24
3.5.2 Logic Programming Languages . . . 24

4 Related Work **27**
4.1 Hibernate Validator . . . 27
4.2 ASP.NET Core MVC . . . 30
4.3 Ruby On Rails . . . 33
4.4 Mendix . . . 35
4.5 OutSystems Forms . . . 36
4.6 Summary . . . 37

5 Technical Approach **39**
5.1 General Overview . . . 39
5.2 Analysis for Validation Rules . . . 40
5.2.1 Data Flow Analysis . . . 41
5.3 OutSystems Model . . . 47
5.3.1 Invariants/Validation Rules . . . 47

6 Implementation **51**
6.1 Data Flow Analysis . . . 51
6.1.1 Extraction of Facts . . . 52
6.1.2 Outputs of the Analysis . . . 57
6.1.3 Generation of Validation Code . . . 58
6.2 Libraries Overview . . . 61
6.2.1 Z3 Theorem Prover . . . 63
6.2.2 SWI-Prolog . . . 63
6.3 Limitations . . . 64

7 Evaluation **67**
7.1 Analysis Overview . . . 67
7.1.1 Objectives . . . 68
7.1.2 Method . . . 68
7.2 Evaluated System . . . 69
7.2.1 Sample Characterisation . . . 70
7.3 Results . . . 70

8 Concluding remarks **75**
8.1 Future Work . . . 76

Bibliography **77**

Appendices 83

A Facts Generated 83

B OutSystems Meta-model 85

C Solutions Obtained 87

List of Figures

1.1 Running Example - Entity Relationship Diagram. 6

2.1 OutSystems Platform - Architecture Overview [44]. 8
2.2 Web Screen Example - OutSystems. 10
2.3 Client Action Example - OutSystems. 10
2.4 Server Action Example - OutSystems. 11
2.5 Entity Example - OutSystems. 12
2.6 Widgets and Built-In Validations - OutSystems. 12
2.7 Server-Side Validation Example - OutSystems. 13

3.1 Control Flow Graph Example. 23
3.2 Path *Sensitive* and Path *Insensitive* Example. 24

4.1 Validation Rule - Mendix. 36
4.2 System architecture - Paweł Krysiak Dissertation [33]. 37

5.1 Mechanism Overview. 40
5.2 Example of an OutSystems action, used as a Control Flow Graph (CFG), (on the left) and an Entity with Validations Rules (on the right). 41
5.3 *Inference Rules.* 44
5.4 Action Flow Example. 46
5.5 Extended Entity View. 49
5.6 Validation Rule View. 49

6.1 Small action used for illustrating the code generation (BEFORE). 59
6.2 Overview of the generated validation code. 60
6.3 Small action used for illustrating the code generation (AFTER). 62

List of Tables

2.1 Relevant OutSystems action elements for our work. 9

4.1 Comparison of related work and our solution. 38
4.2 Comparison of Paweł work and our solution. 38

Listings

3.1 Domain constraint example. . . . 18
3.2 Attribute-Based constraint example. . . . 19
3.3 Tuple-Based constraint example. . . . 19
3.4 Assertion constraint example. . . . 20
3.5 Logic Program Example - Definition. . . . 26
3.6 Logic Program Example - Outputs. . . . 26
4.1 Field constraint example. . . . 28
4.2 Hibernate Validator example. . . . 29
4.3 Implementation of a custom validator in Hibernate Validator. . . . 29
4.4 ASP.NET Core MVC Model with Constraints Example. . . . 31
4.5 Implementation of a custom validation attribute in ASP.NET Core MVC Model. . . . 32
4.6 Validation in the *Ruby on Rails* platform. . . . 33
4.7 Implementation of a custom validation in the *Ruby on Rails* platform. . . . 34
5.1 Example *Rule* facts. . . . 42
5.2 Example *Follow* facts. . . . 43
5.3 Example *FollowsIf* facts. . . . 43
5.4 Example *Assign* facts. . . . 43
5.5 Example *Input* facts. . . . 43
5.6 Example *Node* facts. . . . 43
5.7 Example *Node* facts. . . . 45
5.8 Inference Rule (1) Example. . . . 45
5.9 Inference Rule (4) Example. . . . 46
5.10 Inference Rule (3) Example. . . . 46
5.11 Subset of the latest version of OutSystems Meta-model (before our changes). 48
5.12 OutSystems Meta-model changes. . . . 50
6.1 Example *Rule* facts. . . . 57
6.2 Solutions obtained from the static analysis. . . . 58
A.1 Example *Rule* facts. . . . 83
B.1 Simplification of the OutSystems meta-model. . . . 85
C.1 Solutions obtained from the static analysis. . . . 87

Glossary

Business Rules — *Business Rules* are statements that impose certain constraints on how the business operates. They are specific for each business, and they essentially describe operations, definitions, and constraints that apply to an organization. In a database context, the business rules are used to impose constraints in the values that are expected to be used in the regular operations within the organization.

Acronyms

API	Application Programming Interface
CFG	Control Flow Graph
DbC	Design By Contract
DBMS	Database management system
DFS	Depth First Search
DRY	Don't repeat yourself
DSL	Domain-specific language
IDE	Integrated development environment
MVC	Model-View-Controller
OCL	Object Constraint Language
ORM	Object Relational Mapping
SMT	Satisfiability Modulo Theories
SQL	Structured Query Language

Chapter 1

Introduction

We start this work with a description and contextualisation of the problem, including its motivations, objectives and key contributions. Then we will introduce a running example that will be referenced along in the document. The chapter ends with an overview of the structure of the remaining document.

1.1 Context and Description

Speed and productivity are two of the most critical aspects of the software development industry. Speed is a problem when software development exceeds its time targets and is late to market [4]. Productivity is an issue when the counterbalance between the level of software productivity and the financial costs results in lower profit margins because lower productivity means higher costs [4]. These aspects are critical points to the success of any project or any company.

In engineering projects and more specifically those related to software development, the efficiency and effectiveness of engineers in the software development process, also known as the software development life cycle (SDLC), is crucial. There is an abundance of SDLC models, such as *waterfall*, *spiral*, *unified*, *incremental*, *rapid application development* and *agile* [49]. They differ from each other, but, generally speaking, as a list, they all contain the following stages: analysis and planning, designing, the actual building of the software, testing, deployment, and maintenance operations [49]. Regardless of the approach taken, certain standards apply across the board. Data integrity is one of the standards. The term "data" corresponds to the information collected and used by an application, and the term "integrity" is the property of data consistency and freedom from corruption.

To guarantee that a system maintains data integrity, it is imperative that it must be robust against any data input, whether obtained from the user or another system (such as Application Programming Interface (API) calls). That being said, it is crucial that all the data from external sources should be validated according to established rules.

Usually, there exist rules inherently associated with business policies, that should be considered at all times, and that impose certain constraints on how the business operates. These rules are called *Business Rules*, and they essentially describe operations, definitions, and constraints that apply to an organization. Within these rules, it is possible to find specific information that defines constraints on the data that is expected to be used in the regular operations within the organization. For instance, it may be possible to extract expressions that constrain the values of data, impose conditions on the data type, define the range of values or relationships between attributes, among others. Any failure to prove the invariants must cause the data insertion attempt, whether a specific insertion in the database or any other type of input in the system, to be rejected. When that happens, the system gives feedback about the errors to allow another attempt. It is important to notice the possibility of transient states where not all of the invariants are established. Therefore, it is crucial that, for each layer of the system, there exists an analysis of which constraints should be validated.

This introduces the concept of data validation, that is an essential part of the process of data manipulation. The primary objective is to ensure that, at any given moment in time, all data maintain its integrity, thus ensuring its consistency throughout its entire life cycle and therefore avoiding unexpected problems, such as invalid states or unpredictable errors in the implementation.

Data validation must be employed to any kind of applications, but it is especially sensible in client-server applications. Given the nature of those applications, it is necessary, with the intent to ensure data integrity, to perform validations in all untrusted frontiers of the system. This implies that the data should be validated in the client-side (e.g., when submitting a form) as well as in the server-side (e.g., when processing the data received). It is not advisable to implement just one of the previously referred sides, as it may introduce problems in the system. In the instance were only exists validation on the server-side, the feedback given to the user would only happen after the data has been processed and validated on the server, and the time lost in the data communication to the server can be an issue, possibly leading to negative user experience. By contrast, if there were only validations on the client side, it would be possible to violate those protections and to cause an inconsistent state in the database, consequently causing unexpected errors and possibly security failures. It is common to see client-side input validation implemented in scripts, using for instance JavaScript. The violation of the protections enforced by those scripts is achieved easily by disabling the JavaScript in the browser, meaning that there is a possibility that the validations are not performed in the client side.

1.2 Motivation

Data integrity and its preservation is part of the analysis and application development. Intrinsically, the programmer has the responsibility to define and implement the constraints, but given the invariants nature, it may be necessary to implement them in several layers of the system, possibly introducing code repetition, and increasing the error probability in the duplication of the invariants logic. This is enhanced by the need for the programmer to interpret and understand which invariants validations must take place at any given moment in the system.

Nowadays, there are several ways to define data invariants. One of the most popular and used approaches is related to relational databases and their integrity constraints. It is possible to define and constrain the values that the attributes may take throughout different ways. For instance, one may have an attribute in which the possible values are restricted to some specific content, as it is the case with a gender attribute in an entity representing a user (or a person) where the allowed values are elements of a finite set of possibilities (e.g., male and female). In this particular instance, one can define the constraint directly in the specific attribute. However, the declaration of invariants can also be made over sets of attributes, or by using functions that allow the description of custom constraints and that runs whenever specific conditions arise in the database. This particular approach ensures that the data to be persisted to the database is validated just before the actual creation of the data records, ensuring the last line of defence of data integrity. However, it is not enough to rely only on this approach, since the validation only happens in a late stage of the program's flow. It is crucial to validate the constraints in an early stage, preferably as soon as the data enters the system, since this ensures that a constraint breach is detected as soon as possible, benefiting the user that is quickly notified of inconsistencies in the data inserted. However, if one pretends to ensure the validation of the invariants, that were previously defined in the data layer, in another layer, there is the need to replicate the logic of them manually. As mentioned before, this fact may introduce errors in the interpretation and definition of invariants, and possibly have an adverse effect on productivity.

The existence of a mechanism that through the interpretation of invariants defined in the data model, and according to the possible use of entities at a specific point in the system, materialises in the automatic generation of code that replicates those invariants would be extremely beneficial.

The existence of frameworks and programming platforms such as the OutSystems platform allow for increased productivity and faster application development. The OutSystems platform aims to ensure that the programmer always has the best development experience, assuring that it is as agile as possible and that it is also as intuitive as possible. Contextually, the OutSystems platform provides input validation mechanisms that allow to constraint input fields and to define user logic to enforce extra validation rules. Although, the input fields constraints are minimal, in such a way that they only allow

expressing simple invariants, such as the maximum length of input or the requirement of the input field. It is possible to define custom server-side validations, and with it, the programmer can express specific validation logic of custom constraints (e.g., *Business Rules*). The problem surges when the validation of some rule must happen in a different layer of the application, and no automatic mechanism exists to propagate the written validation code. For instance, one can have some validation logic that runs whenever data enters the server from user input that is pretended to be replicated in the data layer, to ensure the last validation before the persistence of the data to the database. In that instance, or vice-versa, the programmer would need to replicate the validation code manually.

The study carried out by this dissertation intends to offer OutSystems programmers a better process in the definition of invariants and their validation logic. We implemented a mechanism that extends the data model, to permit the definition of constraints and validation logic, allowing to replicate those validations to different layers of the application, whenever exists the possibility to validate data. Therefore, it expects to be an added value for OutSystems programmers and for the platform itself.

This dissertation is built in the scope of the collaboration between NOVA-LINCS[1] and OutSystems. It intends to extend and deepen the work of a previous dissertation done by Paweł Krysiak [33]. That dissertation resulted in a prototype that extended a textual Domain-specific language (DSL), inspired by the OutSystems visual DSL, with constructs that allow describing the validations in a declarative, easy and structured way. The objective was to simplify the way the validation code is added to an application, with the focus on forms, and also to remove the need for the developer to repeat validation code in multiple layers. As referred, the main focus was on the validation of forms, so the solution was built closest to the client side of the application. The programmer was capable of introducing simple constraints, recurring to a declarative approach, such as checking if the given field of a form is empty, as well as more complex ones that may be dependent on multiple fields.

1.3 Objectives

The main objective of this work is the study, analysis, and implementation of a mechanism that allows the programmer to define, at a single point, invariants about the data, and that the corresponding validations are automatically suggested/generated in different layers of the system.

Compared to the dissertation of Paweł Krysiak [33], in this dissertation, the objective is to specify the invariants in the data layer, where the database model is constructed. Then, these specifications will be subsequently replicated and managed as validations in such a way that minimises the repetition of code, in order to maximise the level of

[1] The NOVA Laboratory for Computer Science and Informatics (NOVA LINCS) is a Portuguese leading research unit in the area of Computer Science and Engineering. More details available at: http://nova-lincs.di.fct.unl.pt/.

coverage of data validation with the minimum intervention by the programmer. The determination of the validations and when they are applied is going to be achieved by the use of a data flow analysis technique. In Paweł Krysiak dissertation [33], it was not possible to make data flow analysis *path sensitive* due to time constraints. We ensure that since it is essential to consider the conditions on the program's flow and use that information for the analysis. The proof of concept made by Paweł Krysiak consisted of a prototype that extended a textual DSL, inspired by the OutSystems visual DSL, but its implementation was outside of the OutSystems platform. In this dissertation, the proof of concept is implemented in the platform, namely in the *Service Studio* component (see chapter 2.1.1).

1.4 Key Contributions

At the end of the dissertation, we have a data flow analysis technique capable of propagating data constraints across the various layers of a system. Given a set of invariants defined in the data layer that constrain entities' attributes, our analysis will propagate and manipulate their specification, accordingly to the coded logic. Our analysis gathers all conditional structures expressions, allowing the retrieval of *path sensitive* data invariants.

Additionally, we developed a proof of concept implemented in the OutSystems platform, that uses our analysis and generates code that validates the expressions of the reasoned set of data invariants. By automatically supporting the generation of validation code in each system layer, we remove the programmer's obligation of writing and maintaining it, plus ensuring data integrity and providing faster error feedback to users.

Complementing this dissertation, we present its contributions in a published paper for the INForum, Simpósio de Informática of 2019 [55].

1.5 Running example

As a running example, we present a booking application that is used by hotel staff to book rooms for hotel guests. In the underlying data model of this application, each hotel room accommodates a determined number of adults and number of children. Each room also has a price per night. When a hotel guest calls in to make a room reservation, the booking process begins, and the clerk registers the reservation in the system with the chosen room, the guest's name, the reservation dates, alongside the number of adults and children staying the room. For simplicity purposes, a reservation is constrained to a single room. Figure 1.1 is the entity-relationship diagram of the described example.

In this example, the room entity contains attributes that analysed individually must respect some rules. The price attribute and the adults capacity attribute values are expected to be greater than zero. The children capacity must be greater than or equal to zero. Those invariants also apply to the booking entity, namely for the *NumberOfAdults* and *NumberOfChildren* attributes. It is also possible to define a rule that uses two different

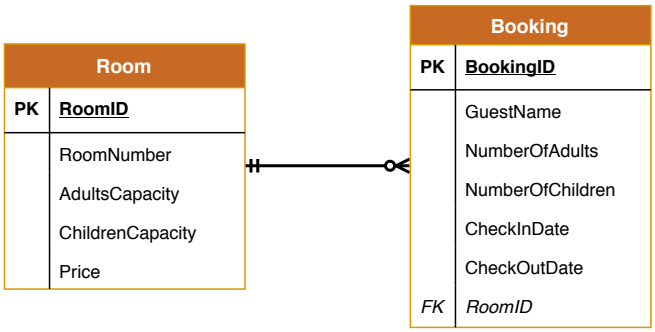


Figure 1.1: Running Example - Entity Relationship Diagram.

attributes of the room entity; for instance, the check-out attribute values must occur on the same day or after as the check-in attribute values. There are also more interesting invariants that should be present, such as the validation that a reservation of a specific room respects its physical capacity. For instance, one of the constraints is the comparison between the attribute *Room.AdultsCapacity* and the attribute *Booking.NumberOfAdults*, wherein the number of adults indicated in a reservation, should not overcome the maximum capacity defined in the room that was assigned to it. This constraint is an example of invariants that involve more than one entity. In this instance, it involves two different entities simultaneously.

1.6 Document Structure

The remainder of this document is organised as follows:

- Chapter 2 - OutSystems Platform: this chapter introduces the OutSystems Platform and some important notions of its Language;
- Chapter 3 - Background: this chapter provides background information on the research that was performed, being the main topic: data flow analysis;
- Chapter 4 - Related Work: this chapter presents related work to this thesis, in this case, focused on the definition and replication of data constraints with the intent to be used in validation(both client and server) and to ensure data integrity;
- Chapter 5 - Technical Approach: this chapter will give a detailed explanation of our data flow analysis technique and the changes made to the OutSystems language;
- Chapter 6 - Implementation: this chapter will give a detailed explanation of our solution's implementation;
- Chapter 7 - Evaluation: this chapter describes how the developed mechanism was evaluated and the results we obtained;
- Chapter 8 - Concluding remarks: this chapter concludes this dissertation with a brief summary of the work done as well as the future work to be done.

Chapter 2

OutSystems Platform

OutSystems is a software company that offers a high-productivity solution using a low-code platform for application development. The OutSystems platform enables visual development of both web and mobile applications, providing an abstraction layer that allows developers to more easily handle the inherent complexity of application development. As a result, significantly faster development times and a higher quality result is achieved when compared to general purpose languages.

2.1 Architecture

The OutSystems Platform architecture [44], which is represented in Figure 2.1, is divided into three main components: *Service Studio*, *Integration Studio* and *Platform Server*.

2.1.1 Service Studio

Service Studio is the Integrated development environment (IDE) that is used to develop applications in OutSystems, and it is characterised as a low-code visual development environment. Server Studio allows defining all layers of an application, such as the data model, user interfaces, logic and business processes, and also asynchronous tasks that will run in the platform [53].

2.1.2 Integration Studio

Integration Studio provides to developers an environment that enables the extension of the OutSystems platform in order to integrate with existing systems, databases and code. Briefly, the Integration Studio allows to fulfil the gap between the traditional development model (e.g., C# code) and the visual model of OutSystems. With the use of extensions [16], it is possible to integrate native technologies such as Microsoft .NET [41], Microsoft SQL

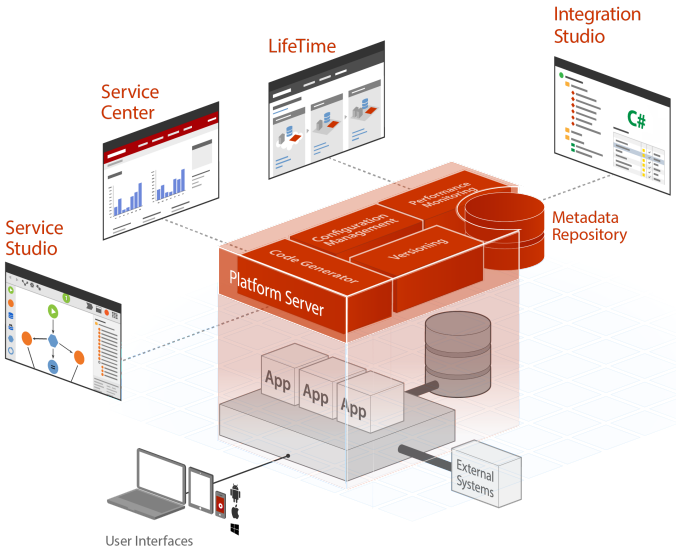


Figure 2.1: OutSystems Platform - Architecture Overview [44].

Server [40], and Oracle Database [43]. After the deployment, the extensions can be later consumed by Service Studio modules for use in web or mobile applications [15].

2.1.3 Platform Server

The Platform Server is the core of the OutSystems platform. It takes care of all the steps required to generate, build, package, and deploy applications, using a set of specialised services [44]:

- *Code generator*: Takes the application modelled in the IDE and generates native .NET [41] code, allowing the generation of applications that are optimised for performance, are secure and run on top of standard technologies.
- *Deployment services*: Deploy the generated .NET [41] application to a standard web application server, ensuring that the application is consistently installed on each front-end of the server farm.
- *Application services*: Manage the execution of scheduled batch jobs, and provides asynchronous logging services to store events like errors, audits, and performance metrics.

2.2 OutSystems Language

In this section, we describe some of the main elements of OutSystems language that are pertinent to this dissertation. Table 2.1 provides a list of relevant OutSystems action

elements that we use in our static analysis, described later in this document.

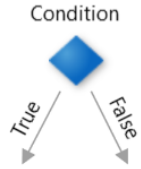
Icon	Name	Description
	If	This node represents the behaviour of an if code block, having a boolean condition and two outgoing branches according to the condition result (true or false).
	Assign	This node assigns a value to a variable. Each assign node can have multiple assignments.
	Server Action	This node invokes an action to be executed on the server-side of the application.
	Entity Action	This node represents an action that manipulates entity records. There exists six different types, such as *CreateRecord*, *GetRecord*, *DeleteRecord*, among others.

Table 2.1: Relevant OutSystems action elements for our work.

In the remaining of this section, we briefly describe the following OutSystems elements: screens, actions, entities. The last section, Validations, describes the possibilities to express validations in OutSystems.

2.2.1 Screens

Screens are the visual interface of the application with which end-users interact. By dragging and dropping objects and their behaviour, also referred to as *Widgets* components, it is possible to design the user interface and get an instant preview of the final result. Each screen may have their actions, input parameters, and local variables. Figure 2.2 illustrates the Service Studio view when designing screens, being visible on the left the *Widgets*, such as *Container*, *Form*, *Input*, *Button*, among others.

2.2.2 Actions

Actions are where the application logic is encoded, and they visually consist of a set of nodes that congregate in a directed graph, or digraph, that represents a given method, function, or procedure. Actions can contain *input/output* parameters and local variables. Action nodes abstractly represent blocks of code, and they can be added to the graph as needed. There exist three different types of actions, namely *Screen*, *Server* and *Client actions*, which are described below. Since OutSystems 11, there is a fourth type of actions called *Service Actions*, but we do not describe them because they are unrelated to this dissertation.

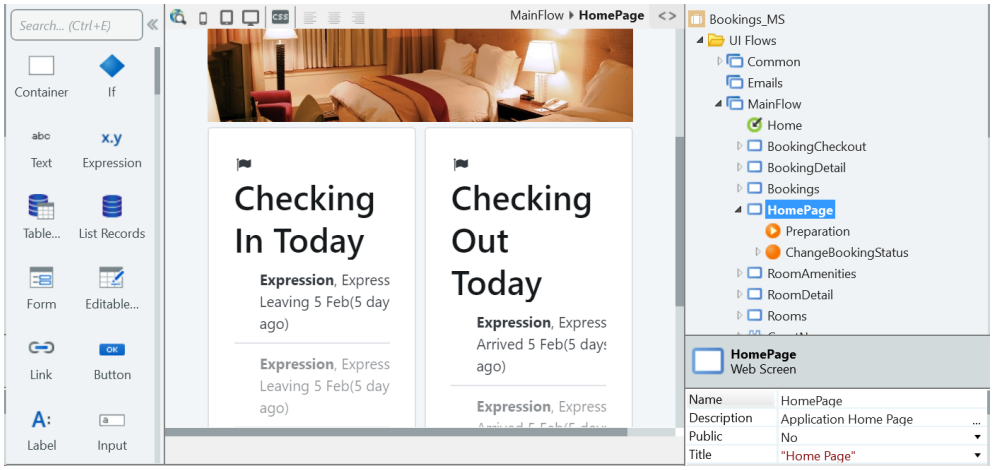


Figure 2.2: Web Screen Example - OutSystems.

2.2.2.1 Screen Actions

A *Screen Action* is an action that is local to a *Screen*, meaning that outside the *Screen*, they are not visible. *Screen Actions* can only be called by *Screen Widgets*, like Buttons or Links. *Screen Actions* are run either on the server (Web applications) or on the device (Mobile applications). For instance, one can define an action that when a user interacts with the screen, such as a click on a particular button or a form submission, it reacts and executes some particular logic. Figure 2.3 illustrates a client action that validates a form, and that outputs a message with an exception whenever errors are detected.

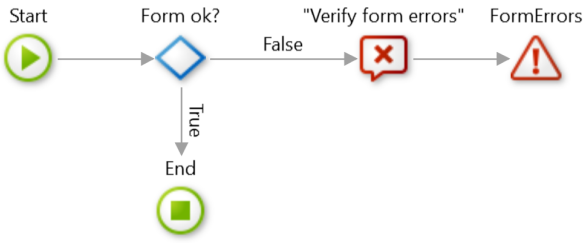


Figure 2.3: Client Action Example - OutSystems.

2.2.2.2 Server Actions

Server Actions encapsulate the logic that implements the business logic of the application that runs on the server-side, both in Web and Mobile applications. For instance, one can define an action that sends multiple emails to the cleaning crew, given a list of dirty rooms, as illustrated in figure 2.4.

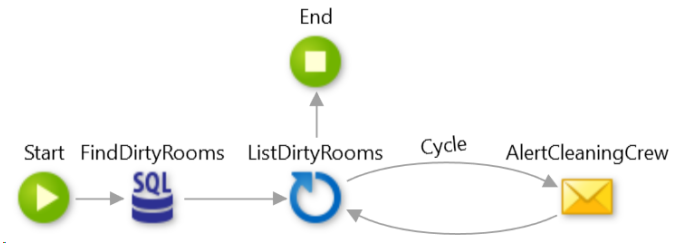


Figure 2.4: Server Action Example - OutSystems.

2.2.2.3 Client Actions

Client Actions[1] are the Mobile equivalent to Server Actions, and they permit to define business logic that runs in the user device, not on a server.

2.2.3 Entities / Database Model

Entities are elements that allow to persist information in the database and to implement the database model. An *Entity* is defined by, among others, a sequence of *Entity Attributes*, and will correspond to a table in a relational database. An *Entity Attribute* is constituted by a name, description, data type, default value and the indication whether it is mandatory. Each of these attributes corresponds to a column in the *Entity's* table. Every *Entity* also has associated operations that allow storage, retrieval, and deletion of individual records from its database table. These operations are called *Entity Actions*. According to the data type, some additional constrains can be placed, such as the maximum length of a text attribute or the indication whether it is an auto number in case of an integer attribute. Figure 2.5 illustrates the *Elements Tree Area* of the *Data layer* and the *Properties Area*, both from the *Service Studio* component. In the elements tree area, it shows an entity with five attributes (*Id*, *RoomNumber*, *AdultsCapacity*, *ChildrenCapacity*, *Price*) and also the *Entity Actions* (orange icons) that were generated for this specific entity. In the Properties Area is shown the properties for a particular entity attribute, in this case, the properties of the *AdultsCapacity* attribute.

2.2.4 Validations

The OutSystems platform provides input validations mechanisms, that can be used to validate data from user input, such as web forms, and they can be split into two types: *Built-in validations* and *Custom Server-side Validations*. The first mechanism is directly associated with forms for user input, and the validations are specific for each input element. The second mechanism is more generic and customizable, and it can be used for the data validation that arrives from forms but also to any other procedure that requires custom data validation. We describe both mechanisms below.

[1] In the current version of the OutSystems Service Studio 11.0, the Client Actions can only be defined in the development of mobile applications.

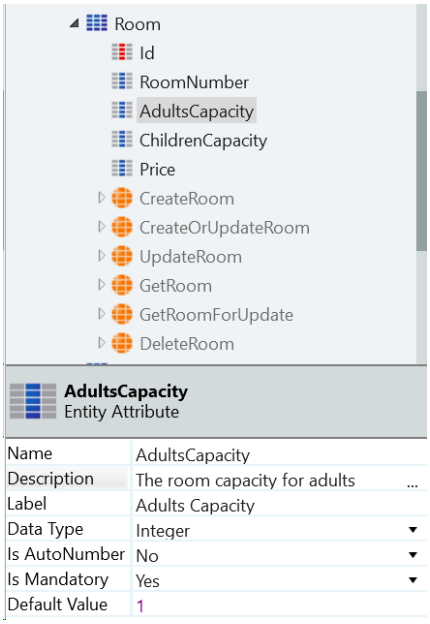


Figure 2.5: Entity Example - OutSystems.

2.2.4.1 Built-In Validations

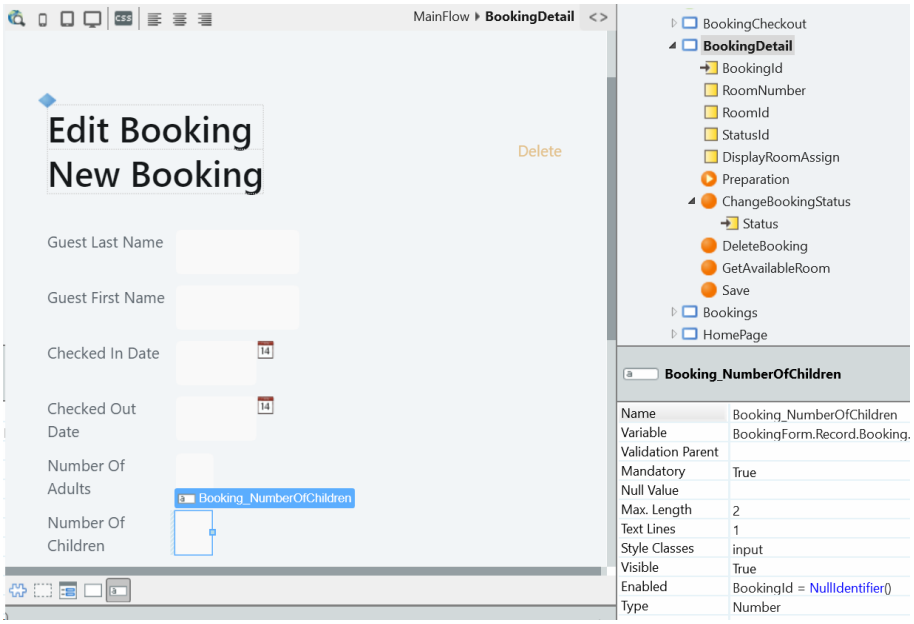


Figure 2.6: Widgets and Built-In Validations - OutSystems.

The built-in validations consist in two different validations: *Mandatory* and *Data Type*. The *Mandatory* validation is run whenever any input element is marked as mandatory, meaning that the user must type in a value. The *Data Type* validation checks if the value typed complies with the data type of the variable bound to the input element. This validation also allows expressing simple invariants, such as the maximum length of the input element. As stated before, this type of validations is associated with input elements. The

figure 2.6 illustrates a form in a Web Screen in which the selected input element (*Booking_NumberOfChildren*) has the properties shown in the *Properties Area*, where it is possible to define the constraints that enable the built-in Validations (*Mandatory*, *Max.Length*, *Type*).

2.2.4.2 Custom Server-Side Validations

The custom server-side validations allow writing user logic that enforces extra validation rules. It can express conditional statements, perform database queries, among others. Figure 2.7 is an example of server-side custom validation that imposes constraints to the data that comes from user input before invoking the function that creates a new record in the database. In this specific figure, we are validating, among others, if the number of adults indicated in a reservation does not overpass the physical limit of the room selected. This type of validation requires information that it is only available on the database (e.g., the Room data) that is obtained throughout *Entity Actions* (e.g., GetRoom action).

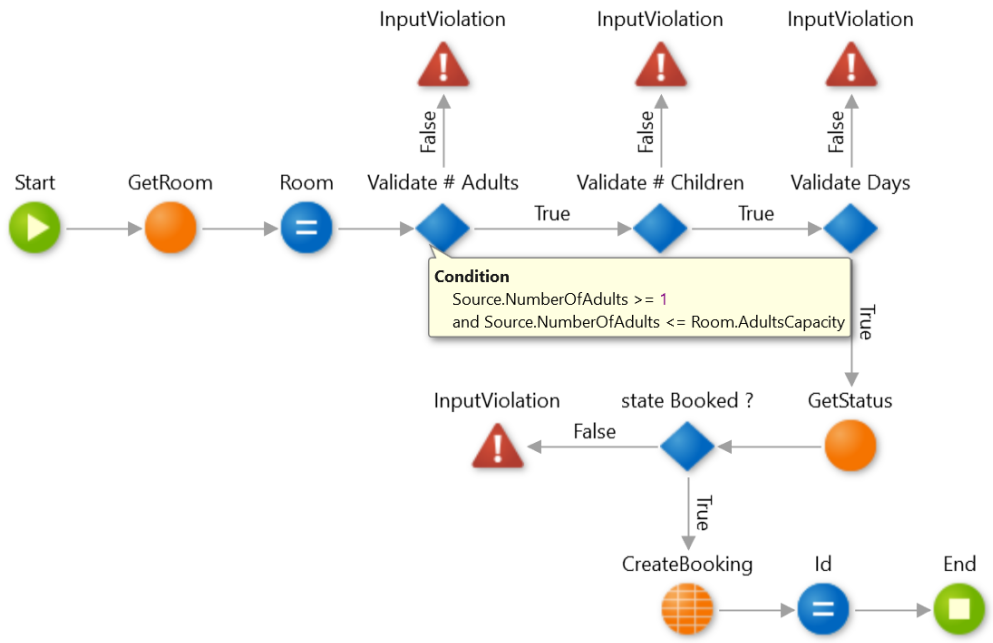


Figure 2.7: Server-Side Validation Example - OutSystems.

As described in chapter 1.2, there are some limitations in the OutSystems platform regarding validations. In this chapter, it was described some of the possibilities to define validations in the platform, such as the simple widget validations(*Mandatory*, *Data type*, *Length*) in the user interface, but also the custom server-side validations.

Chapter 3

Background

This chapter presents background context related to this dissertation, including Design by Contract, Refinement Types, Logic Programming Languages and Data Flow Analysis.

3.1 Defensive Programming

Defensive programming is a software engineering practice that aims to ensure that a given application behaves consistently and deterministically even under unforeseen circumstances [37, 39]. One of the main rules of defensive programming is to never assume anything about the inputs or the state of a program, meaning that, we should always assume the worst case scenario. In particular, it encourages programmers to verify the inputs and the programs' states by including as many verifications as possible, even if they are redundant. This approach introduces, however, runtime overhead and contributes to the software's complexity, affecting software reliability and maintainability. For instance, if there is the need for the change of a specific verification routine and if it is replicated in multiple locations throughout the system, it becomes difficult to maintain them consistent with one another.

In our approach, we will attempt to improve the process of data validation, ensuring that the validations are performed at different layers, whenever needed, without the obligation of duplicated code implicit made by the programmer, and guarantee that the programmer needs to write the validation code only once.

3.2 Design By Contract

Design By Contract (DbC) [38, 39] is an approach to design software that defines how elements of software should collaborate with each other on the basis of mutual *obligations*

and *benefits* [21], with the intent to ensure the reliability of the software, mainly *correctness* and *robustness* [39]. It has been developed in the context of object-oriented programming, it is the basis of the programming language Eiffel [14], and it is well suited to design component-based and agent systems [21].

To ensure *correctness*, DbC relies on assertions, which are used to define the semantic specification of routines. The conditions expressed by assertions are boolean expressions. Whenever a routine is defined, it must be specified two assertions associated with it: a *precondition* and *postcondition*. The precondition states the properties that must hold whenever the routine executes, and the postcondition states the properties that routine guarantees when it returns, if and only if the precondition was fulfilled. A precondition violation indicates a breach of the contract in the client(caller), meaning that the caller did not respect the conditions imposed by the supplier(routine). A postcondition violation indicates a breach of the contract in the supplier(routine), implying that the routine failed to ensure a correct state after execution.

In contrast with defensive programming, it is possible to avoid redundant tests and validations. This approach replaces the need for verifying and protect every possible state by assigning the responsibility of each consistent condition to one of the parties. Additionally, if the contract is precise and explicit, there is no need for redundant checks.

In our approach, we are interested in ensuring, by means of runtime verification, that the client(caller) respects the conditions/assertions before calling the supplier (in this case the server) because we can assume that the supplier(server) ensures the postconditions defined by the programmer, if and only if the logic is correctly programmed.

3.2.1 Class Invariants

A *class invariant* is a property that is used to express which states of the objects of the class are consistent, transcending particular routines [27, 38]. This is an extension of the notion of *representation invariant* since those invariants are conditions on the state of an object that are true in all observable states.

As stated before, preconditions and postconditions define the conditions of an individual routine, but class invariants have the intention to ensure more global properties of the instances of a class, in such a way that they apply to all contracts (routines) of a certain category [38].
Bertrand Meyer characterise a class invariant with the following properties: [38]:

- "The invariant must be satisfied after the creation of every instance of the class. That means that every creation procedure of the class must yield an object satisfying the invariant".

- "Every exported routine of the class must preserve the invariant (that is to say, every routine available to clients). Any such routine must guarantee that the invariant is satisfied on exit if it was satisfied in the entry".

As stated before, a class invariant characterises acceptable and consistent states, so there may exist intermediate states that do not satisfy the invariant. Most of the computations do violate the invariant, such as new value assignments, but it is acceptable in this context, as long as at the end of a specific routine the invariant is re-establish. This is a valuable property that our work pretend to guarantee, since in the context of validation, and more explicitly in this dissertation topic, the main focus is to verify and ensure that the objects states are valid whenever exists data entering in the application. Also so when that data is persisted to the database, allowing that in the "middle" the invariants may fail to be satisfied.

3.3 Relational Databases - Integrity restrictions

Data integrity refers to the consistency, accuracy, and reliability of data stored in a database. Therefore, when designing databases, it is crucial to pay attention to how to support and maintain data integrity. To enforce data integrity, one can constrain or restrict the data values that users are allowed to pass when inserting new records or updating existing ones. There are different types of restrictions that can be used. Edgar Codd, who proposed the *relational model* for database management [10], introduced five types of integrity restrictions that can be applied in a database context [10]: *Domain integrity*, *Column integrity*, *Entity integrity*, *Referential integrity*, and *User-defined integrity*. Entity integrity and referential integrity apply to the relations in every relational database, with *entity integrity* being associated with *primary keys* and *referential integrity* to *foreign keys*. Domain and column constraints set the allowed values for a given attribute or set of attributes. User-defined constraints define actions that occur whenever the condition tested does not respect the specification.

Entity constraints and referential constraints are out of the scope of this dissertation since there is no need to establish new restrictions or update existing ones within their domain (*primary keys* and *foreign keys*). User-defined constraints involve *stored procedures* [20, 54], *triggers* [20, 54], *batches* [54] or any other user-defined function. In this context, only *triggers* are described since it allows us to define functions that are run when specified actions occur within a database. We will also introduce the concept of *database assertions*, which are constraints that are used to restrict attributes in more than one table and are always satisfied by the database [20].

The following sections 3.3.1 and 3.3.2 presents more details about domain constraints and column constraints, respectively. Section 3.3.3 will introduce assertions. For last, section 3.3.4 describes triggers in more detail. In all those sections, some examples are presented in Structured Query Language (SQL), that is a standard language for accessing and manipulating databases. SQL provides a set of methods for defining *integrity constraints* that are assured to be enforced by the SQL-implementation.

3.3.1 Domain Constraints

A *domain constraint* specifies the set of possible values that an attribute may have, and it is essentially a data type with enforced invariants [10]. Domain definitions are particularly useful when several tables contain identical column definitions. Instead of repeated restrictions trough multiple tables, the domain is defined only once and used when necessary. When the restrictions are associated with a distinct column in a specific table, they are usually called *column constraints*. Section 3.3.2 describes all the possibilities within this type. The form of *domain constraints* is the following:

```
CREATE DOMAIN <domain-name> AS <data-type> CHECK (<conditional-expression>)
```

For instance, one can define a domain called *POSITIVE_INT*, that extends the data type *INT*, with the constraint that only integer values greater than or equal to zero are allowed, as shown in the listening 3.1. With this, it is possible to use the domain when creating new tables, or updating existing ones, without being restricted to a specific column or table. The domain *POSITIVE_INT* is, in fact, a refinement type (see chapter 3.4), although in the context of databases its condition is validated at runtime and not at compile time.

```
1  CREATE DOMAIN POSITIVE_INT AS INT CHECK (value >= 0);
2  CREATE DOMAIN INTERVAL_REAL AS REAL CHECK (value >= 0.0 AND value <= 1000.0);
3
4  CREATE TABLE room
5    (
6      id INT,
7      roomnumber POSITIVE_INT,
8      adultscapacity POSITIVE_INT,
9      childrencapacity POSITIVE_INT,
10     price INTERVAL_REAL
11   );
```

Listing 3.1: Domain constraint example.

3.3.2 Column Constraints

As referred before in section 3.3, *column constraints* are specific for a given attribute or set of attributes of a particular table [10]. One reason that *column constraint* is part of the relational model is that it makes possible to abstain the complexities and proliferation of domains that are subsets of other domains [10]. So with the usage of *column constraints*, one can extend a *domain constraint* to specify an additional range constraint, for instance, consider the definition of the currency type EURO(€) as a *domain constraint* that can be later used to constrain the values of rooms prices.

There exist two specifications of column constraints [20]: one constraining a single column/attribute, called *attribute-based*; and another constraining multiple columns/attributes in a table, called *tuple-based*. Both are described in section 3.3.2.1 and 3.3.2.2, respectively.

3.3.2.1 Attribute-Based

An *attribute-based constraint* is a condition associated with a single attribute that must hold in each tuple of its relation/table [20]. It is possible to involve other attributes and relations, but only in subqueries, and the condition is evaluated whenever the associated attribute changes, namely through operations of insertion and update [20]. The representation of *attribute-based constraints* is the following:

```
CHECK (<conditional-expression>)
```

An example of this constraint is listing 3.2, with the constraining of the attributes values of the table Room.

```
1  CREATE TABLE Room
2      (
3          id INT,
4          roomNumber INT
5                 CHECK ( roomNumber IN SELECT number FROM roomsNumbers ),
6          adultsCapacity INT
7                 CHECK ( adultsCapacity > 0 ),
8          childrenCapacity INT
9                 CHECK ( childrenCapacity >= 0 ),
10         price REAL
11                CHECK ( price >= 10.00 AND price <= 1000.00 )
12     );
```

Listing 3.2: Attribute-Based constraint example.

3.3.2.2 Tuple-Based

A tuple-based constraint is similar to the invariant described in 3.3.2.1 (*attribute-based*). The representation is similar but is defined as a separate element of table declaration, making it possible to refer to any attribute of the relation, but other relations/attributes require a subquery. Whenever the tuple is inserted or updated, the conditions of the constraint are evaluated [20]. For instance, one can define a constraint that restricts the maximum price of all the rooms, but also opens an exception for a specific room, as demonstrated in listing 3.3.

```
1  CREATE TABLE Room
2      (
3          id INT,
4          roomNumber INT,
5          adultsCapacity INT,
6          childrenCapacity INT,
7          price REAL,
8          CHECK ( (roomNumber = 1) OR (price <= 1000.00) )
9      );
```

Listing 3.3: Tuple-Based constraint example.

3.3.3 Schema-Level Constraints - *Assertions*

Schema-Level constraints or *assertions* are database-schema constraints that express a condition that is intended to be always satisfied by the database [20]. Whenever the tuple is inserted or updated, the conditions of the constraint are evaluated, with the possibility to refer to any attribute or relation in the database schema. [20]. The representation of *assertions* statements in SQL is the following:

```
CREATE ASSERTION <name> CHECK (<conditional-expression>)
```

As it is broader than the previous constraints, it is easier to represent constraints that correlate multiple relations and their attributes.

```
CREATE ASSERTION booking-persons-constraint CHECK (
    NOT EXISTS (
        SELECT *
        FROM Room r,
               Booking b
        WHERE r.id = b.roomId
        AND (
               b.numberOfAdults > r.adultsCapacity
            OR b.numberOfChildren > r.childrenCapacity
            )
    )
);
```

Listing 3.4: Assertion constraint example.

For instance, one pretends to ensure that whenever exists a new booking for a particular room the occupancy of adults and children should not overcome the physical occupancy of the room. This example can be ensured by creating an assertion that validates that does not exist a book where the occupancy is bigger than the one physically possible, as illustrated in listing 3.4. Schema-Level constraints can be used to express *Business Rules* that are meant to be guaranteed always by the database. This way, it provides the last layer of validation before the actual data is saved in the database.

3.3.4 User-Defined Constraints - *Triggers*

User-Defined constraints extend the previously described integrity constraints allowing to define specific statements, in a way that can be enforced by the Database management system (DBMS) [10]. Once these constraints are defined, DBMS enforces them, and consequently, it is not dependent on voluntary compliance by programmers or end users [10]. The prime objective of this constraints, as also the others integrity constraints, is to assure that the database keeps an accurate state by preventing violations, but user-defined constraints also trigger specified actions when specific conditions arise in the database [10]. These constraints can be defined and expressed, for instance, by the usage of databases

triggers. *Triggers* are only one possible way, but there exist other possibilities as enumerated before in section 3.3. A database trigger is a set of SQL statements that are run when specified actions occur within a database.
To design a trigger mechanism, two requirements must be fulfilled [54]: 1. Specification of when a trigger is going be executed, including the *event* that causes the trigger to be checked and the *condition* to be evaluated for the trigger execution to proceed. 2. Specification of the actions to be done when the trigger executes.
Triggers can be executed *before* or *after* operations(*events*) of the SQL statements `INSERT`, `UPDATE` or `DELETE`. The conditions to be evaluated are syntactically boolean expressions.

The implementation of triggers depends a lot of vendor-specific details, so differences exist for each DBMS that includes it. Triggers appeared officially in the SQL:1990 standard, and the simplest representation is:

```
CREATE TRIGGER <trigger-name> [ BEFORE | AFTER ] <event> ON <table-name>
WHEN (<condition>)<statement list>
```

The usage of triggers to express *Business Rules* that got the intent to constraint data values is not directly applicable. Although it can be used to express other type of *Business Rules*, for instance, when a person ends its staying, an email is sent to the cleaning services to clean the room to the next person. Triggers can also be used to generate derived column values, event logging or synchronous replication of tables, among others [20].

All of the previously described possibilities to define integrity restrictions in relational databases are verified in run time. The use of those restrictions ensures the last line of defence of data integrity. In our approach, it is essential that the validation is made at run time since it is when the data is available from external sources of the system. It is also crucial to ensure the last validation before the persistence of the data to the database, and that is possible with integrity restrictions in the database.

3.4 Type Systems and Refinement Types

A Type System is formulated as a set of rules for checking the consistency of programs through static analysis [46]. The primary purpose is to prevent the occurrence of errors that lead to illegal program states [5]. Type systems are used to organise compilers (including compilation and optimisation), model languages or structure information, among others [5]. It relies on the definition and usage of constraints called *types*. *Types* are assigned to variables, and it constrains the possible values during every run of a program [5]. If `x` has type Boolean, it is expected that the variable assumes only boolean values. Languages, where variables are restricted using types, are called typed languages [5].

Refinement types are *types* enriched with logical predicates which are assumed to hold for any element [34, 61]. For instance, one can define the refinement type *Nat*, which is intended to represent the natural numbers, corresponding to the set of `Int` values which additionally satisfy the logical predicate of values greater than or equal to zero.

This example can be defined by the following notation:

$$type\ Nat\ =\ \{\ n : Int \mid n\ >=\ 0\ \}$$

Refinement types can also be used to express both preconditions and postconditions of a function, when used as function arguments and return types, respectively [61]. This means that we can define any function type by specifying contracts. For instance, one can define a function that receives natural numbers as input, and outputs natural numbers greater than 10, as shown below:

$$function\ f\ :\ Nat\ \rightarrow\ \{\ n : Nat \mid n\ >\ 10\}$$

There are multiple implementations of this concept on programming languages, such as Haskell [60], Meta Language (ML) [19] and TypeScript [62].

In our approach, the notion of refinement types can be useful to express the invariants of specific attributes within an object or to define the preconditions and postconditions of functions. However, it is difficult to express more complex predicates, but the main disadvantage is that the predicate of the refinement type must be respected in all possible states, which impossibilities intermediate states. As the data insertion and validation mainly occurs at runtime, it may exist states where the predicate is violated. Refinement types purpose is to ensure always the invariant defined by the type systems run.

3.5 Data Flow Analysis

In previous chapters, the main topic is relative to data integrity, including practices and approaches with the objective to express constraints. This chapter presents a technique, named data flow analysis, that can be used to analyse the data flow and to differ where the validation of the constraints should be applied.

Data flow analysis is a static analysis technique that derives information about the useful properties of programs being analysed. It focuses on computing information, that is guaranteed to hold at any execution, for every single program point [30, 45, 50]. With this information, it is possible to do certain inferences about the use of variables or expressions, for instance, to discover variable values that aren't used anymore or to know which assignments defined the current values of variables.

Data flow analysis is usually performed on the program's control-flow graph (CFG). *A CFG is a directed graph in which the nodes represent basic blocks and the edges represent control flow paths* [1]. Each basic block (node) consists on a linear sequence of program instructions having one entry and one exit point. As it is a graph, each basic block (node) may have many predecessor and many successors, also with the possibility of one be its own successor. There is a control flow path(edge) from block *b1* to block *b2* if the control may flow from the last program instruction in *b1* to the first program instruction in *b2*. Figure 3.1 represents the control flow diagram of a simple function that validates one variable.

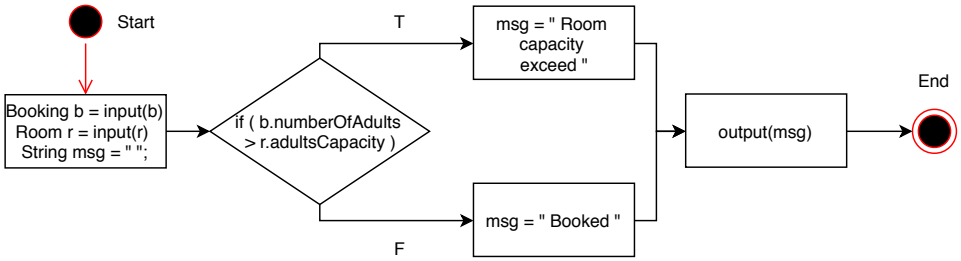


Figure 3.1: Control Flow Graph Example.

The term "data flow analysis" can be split into *local*, *global* (or *intraprocedural*) and *interprocedural* data flow analysis [30]. *Local* data flow analysis is performed for a *basic block* only once. *Global* data flow analysis consists in the analysis over *basic blocks* in a CFG, that may require repeated traversals but always confined to a function/procedure. *Interprocedural* data flow analysis is when the analysis is performed across several methods (and classes) or procedures.

Data flow analysis is typically *flow sensitive*, meaning that it takes into account the order of program instructions(statements), and also is *path insensitive* (see chapter 3.5.1). Specifically for *interprocedural* analyses, it is also possible to define the *context sensitivity*. *Context sensitive* means that each method/procedure call is aware of the callee and the output is correctly retributed to the callee. In contrast, *context insensitive* may lead to situations where the output of a method/procedure call may be returned to a different callee. Fully detailed data flow analysis is usually obtained in *intraprocedural* analyses. Analyses that span whole programs (*interprocedural* analyses) use techniques that discard or summarise some information, and they are usually *flow-insensitive* and employ a limited amount of *context sensitivity* [65].

The following distinctions characterise data flow analyses *flow-sensitive* [50]:

- **Forward analysis** - In this analysis, the information at a node depends on what happens earlier in the flow graph. *Available expressions analysis* is an example of forward analysis that aims to discover the expressions whose results at a specific node are the same as their previous computed values despite the path taken to reach the specific node [30].

- **Backward analysis** - In this analysis, the information at a node depends on what happens later in the flow graph. *Liveness analysis* is an example of backwards analysis that determines whether the value of some variable, at a specific point, is going to be used in the future [17, 30].

We can narrow even further the characterisation of data flow analyses, by restricting the type of the information obtained, as follows [50]:

- **May analysis** : In this analysis, the information described at a node **may** possibly be true. Examples of this are *live variables analysis* and *reaching definitions analysis* ;

- **Must analysis** : In this analysis, the information described at a node **must** definitely be true. Examples of this are *available expressions analysis* and *very busy analysis*.

3.5.1 Path-Sensitive Data Flow Analysis

As stated before, data flow analysis is typically *path insensitive*, resulting in loss of information since no track of eventual conditions exists in the program flow. Figure 3.2 illustrates the difference between a path *sensitive* and *insensitive*. Although as it may

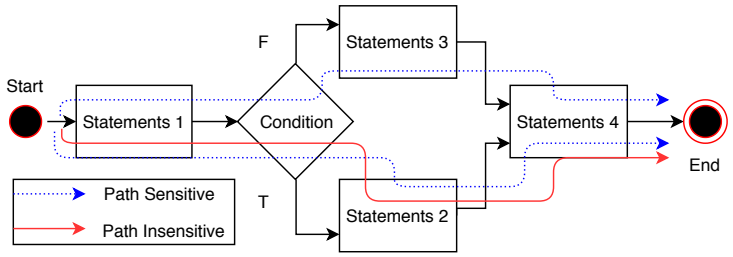


Figure 3.2: Path *Sensitive* and Path *Insensitive* Example.

appear to be a problem, it is possible to make a *path-sensitive* analysis by tracking facts for each possible branch in the program flow. This should be done carefully and always paying attention to the complexity since it may result in an exponential or infinite search space [11].

The usage of data flow analysis is a must considering that one of the objectives of this dissertation work is to provide "tips" to the programmer about the validations that should be established in other layers(such as in client-side) regarding the invariants defined before in the data layer. To ensure that, some data flow analysis must be done to verify which entities and their attributes are going to be manipulated, for instance, on forms, and decide which of the invariants should be replicated. With this, the primary objective is to detect invalid data in an earlier point of the program flow, without the need of the programmer repeat the constraints logic that was already defined.

The data flow algorithms are typically implemented using a fixed-point iterative algorithm [31, 42]. However, we decided to pursue another approach. Yannis Smaragdakis [56] proposed the use of Datalog, a logic programming language, also referred by Frank Pfenning [45] in his lectures notes on compiler design. The formulation of the various properties to analyse and their propagation rules, using Datalog, is very convenient, as it is easy to write and understand the results. In the next section, we will define logic programming languages, and also describe the differences between Datalog and Prolog.

3.5.2 Logic Programming Languages

Logic Programming Languages, or Declarative Languages, are described in the form of symbolic logic, and they produce results by using a logical inferencing process [51]. In *procedural* programming languages, such as C or Java, the programmer must specify in

detail ***how*** to solve a problem. However, in declarative programming languages, the programmer focus on stating ***what*** is the goal to be achieved and the system will try to find a solution. The programs written in a logic programming language consist of a set of statements, or propositions, in a logical form, expressing relations represented as facts and rules [7]. The knowledge base, necessary to derive solutions, is built on a set of predicates, from which facts can be defined, and by rules, which describe relationships between predicates and allow us to infer new facts from previously facts. It is necessary to perform a query, or question, to obtain the program solution. It validates whether a specific relationship can be derived by considering the knowledge base. Rules are written in the form of clauses

$$H :- B_1, ..., B_n \qquad [7]$$

and are read declaratively as logical implications [7]: assume that, if B_1 and ... and B_n all hold, then H also holds. H is called the head of the rule, and $B_1, ..., B_n$ is labeled the body. Facts are expressed similar to rules, but without a body. For instance, the fact "It exists a railway connection between Lisboa and Porto" can be represented as:

$$connection(Lisboa, Porto).$$

Each item in the parenthesised list following the name of the fact is called a term. A term is either a constant(atom or number), a variable or a compound term. The rule "If exists a railway connection between X and Y and, if also exists a railway connection between Y and Z, then it also exists a railway connection between X and Z" can be represented as:

$$connection(X, Z) :- connection(X, Y),\ connection(Y, Z).$$

Prolog and Datalog

Prolog and Datalog are two examples of logic programming languages. From the syntactical point of view, Datalog is a subset of Prolog; hence, each set of Datalog clauses could be parsed and executed by a Prolog interpreter [7, 8]. In Datalog, the terms are restricted to being either variables or constants; in Prolog, the terms can also be compound terms. Prolog produce answers with a *tuple-a-time* approach, computed with the *depth-first search* strategy, whereas Datalog uses a *breadth-first search* strategy, which produces the set of all solutions [7, 8]. One of the main problems with Prolog is its termination behaviour. The termination of a recursive Prolog program depends strongly on two points. The first one is the order of the rules in the program. The second one is the order of the literals defined in the rules. From a Datalog viewpoint, the order of clauses and literals is totally irrelevant [7, 8].

As an example of this issue, consider the following program [58], including its facts, rules and a query:

```
// Facts
connection(amsterdam, schiphol).
connection(amsterdam, haarlem).
connection(schiphol, leiden).
connection(haarlem, leiden).

// Rules
connection(X, Y) :- connection(X, Z), connection(Z, Y).
connection(X, Y) :- connection(Y, X).

// Query
?- connection(amsterdam, X).
```

Listing 3.5: Logic Program Example - Definition.

The program in listing 3.5 is syntactically correct either in Datalog or Prolog. However, the behaviour and, ultimately, the results are different in both languages. If one executes this program using a Datalog interpreter, it produces the correct expected answer, as shown in listing 3.6. However, using a Prolog interpreter, it loops forever. This behaviour happens due to the differences that were stated before, meaning that a Prolog programmer should avoid writing looping programs, while it should not be a problem to a Datalog programmer.

```
?- connection(amsterdam, X).
X = amsterdam ;
X = haarlem ;
X = schiphol ;
X = leiden.
```

Listing 3.6: Logic Program Example - Outputs.

Chapter 4

Related Work

In the context of this thesis, the research upon the related work will be focused on the existing mechanisms, tools, and frameworks that allow a programmer to define custom data constraints in a single location of the system, and is expected that those invariants are replicated and used whenever necessary in different layers of the application, including server-side and client-side.

There are many software pieces that in one way or another include some aspects that our work pretends to study. The reason behind the chosen software tools described as related work relies on their validation features, as well on their simplicity to express data invariants.

4.1 Hibernate Validator

Hibernate is a Object Relational Mapping (ORM) solution for Java [29] environments [12]. ORM is a term that refers to the technique of mapping data between relational databases and object oriented programming languages [12, 18]. Hibernate takes care of the mapping between Java classes and database tables, and also from any Java data type to SQL data types. With it, the development time can be decreased significantly that otherwise would be spent with manual data handling in SQL [12].

Hibernate Validator is the reference implementation of Bean Validation, which is a Java specification that standardises constraint definition, declaration and validation [3, 13]. Bean Validation 2.0 is the current version, that is defined by the JSR 380[1], and implemented by Hibernate Validator 6.0. It is important to notice that Hibernate Validator can be used independently of the usage of Hibernate ORM.

[1] Java Specification Requests (JSRs) are the actual descriptions of proposed and final specifications for the Java platform. JSR 380 details are available at: https://jcp.org/en/jsr/detail?id=380.

Validations

Hibernate Validator permits to express constraints, via Java annotations, that enhance the object model and empower the validation of it. This is possible by defining the referred constraints within four levels: *Field Constraints*, *Property Constraints*, *Container Element Constraints* and *Class Constraints*. Although, it is not possible to place all annotations on all of these levels, such as in the class constraints level.

Hibernate Validator supports all the constraints defined in the Bean Validation API, but also extends it with more custom and complex annotations. As for the basic constraints, some annotations are: *@AssertFalse*, *@Max(value=)*, *@NotNull*, *@Pattern(regex=, flag=)*, *@Size(min=,max=)*. Within the extended assertions, it provides for instance: *@CreditCardNumber*, *@Email*, *@NotEmpty*, *@Range(min=,max=)*. Although, in cases that the provided annotations are not sufficient, it is possible to create custom constraints that are closest to more specific validation requirements. This will be addressed next.

Before discussing how the validation is done, it is important to understand how the constraints are imposed. For instance, if one intends to introduce a validation on a given field, the way to do it is by denoting an annotation, as shown in listing 4.1.

```
1  public class Room {
2      @NotNull
3      private int id;
4
5      @NotNull
6      private int roomNumber;
7
8      @NotNull
9      @Range(min = 1, max = 10)
10     private int adultsCapacity;
11
12     @NotNull
13     @Range(min = 0, max = 5)
14     private int childrenCapacity;
15
16     @NotNull
17     @Positive
18     private double price;
19     // constructors, getters and setters...
20 }
```

Listing 4.1: Field constraint example.

After declaring the constraints, with the use of annotations, the process to perform a validation of these constraints requires a *Validator* object. A *Validator* instance is thread-safe and may be reused multiple times. To discover which constraints violations an object may have, it is used the `validate()` method that returns a set of *ConstraintViolation* instances. Those instances can be iterated over in order to see which validation errors occurred. This is illustrated in listing 4.2.

```
ValidatorFactory factory = Validation.buildDefaultValidatorFactory();
validator = factory.getValidator();

// Room(id,roomNumber, adultsCapacity, childrenCapacity, price)
Room room = new Room(1, 1, -1, -10.0);

Set < ConstraintViolation < Room >> constraintViolations =
validator.validate(room);

if (constraintViolations.size() == 0) {
    all ok...
} else {
    errors... (This object has some constraints violations !)
}
```

Listing 4.2: Hibernate Validator example.

The usage of the basic assertions or even the complementary assertions provided by Hibernate Validator may be useful in common situations, but in most cases there is the need to define a more explicit constraint and validation. The concept of custom constraints is interesting since it allows to define *Business Rules* and to define the logic behind the validation. In order to define a custom constraint, one needs to create a constraint annotation and to implement a validator. After the creation of the constraint annotation (that is not illustrated in this document) and the implementation of the validator (listing 4.3), one can use it, for example, as a *Class Constraint* to constraint the instances of objects of a specific class.

```
public class ConsistentRoomCapacityValidator implements
ConstraintValidator < ConsistentRoomCapacity, Object > {

    @Override
    public boolean isValid(Object value, ConstraintValidatorContext context) {

        if (!(value instanceof Room)) {
            throw new IllegalArgumentException(
                ``Illegal method signature, expected type Room.''
            );
        }

        return ((Room) value).getAdultsCapacity > 0 &&
        ((Room) value).getChildrenCapacity >= 0;
    }
}
```

Listing 4.3: Implementation of a custom validator in Hibernate Validator.

Analysis

With the possibility to define constraints directly in classes, it guarantees that there is a unique place where the data conditions are defined, but also ensures that there is no need to replicate them over the system. With an instance of a *Validator* object, it is possible to validate the fields of an object whenever necessary. The capability to define custom constraints is an important feature, and the fact that it allows introducing invariants that constrain objects of a different kind is a valuable component. Also, if the programmer combines Hibernate Validator with Hibernate ORM, it ensures that whenever the object transits from transient to persistent state, the validation of the constraints happens, ensuring an additional layer of data integrity. However, Hibernate Validator and Hibernate ORM are server-side frameworks, and this does not apply entirely to our problem, as we aim for both server-side and client-side validations. Even though Hibernate Validator allows to define custom constraints and it avoids breaking the Don't repeat yourself (DRY) principle [28], it is up to the programmer to explicitly invoke the proper function that validates the specific object instance, whenever necessary. If the programmer misses to invoke the function, and in the case that only Hibernate Validator is used, there is no automatically validation mechanism meaning that, in last resort, invalid data is only detected in the database.

4.2 ASP.NET Core MVC

ASP.NET is an open source, server-side web application framework created by Microsoft that runs on Windows used for building modern web apps and services with .NET [2, 41].

ASP.NET Core is a redesign of ASP.NET 4.x and is an open-source, cross-platform framework for building modern cloud-based web applications on Windows, MacOS, or Linux [2]. One of the main differences between both ASP.NET and ASP.NET Core is that the first is build for Windows and the second provides the ability to run the applications (web/service) on multiple platforms (Windows, Linux, and macOS), using .NET Core [2].

One of the features that integrate ASP.NET Core is the Model-View-Controller (MVC) architectural pattern [18, 32]. **ASP.NET Core MVC** is how it is recognized, and it includes, for instance, routing, model binding, model validation, web APIs, strongly typed views, view components, and some more. The most exciting feature for our analysis is the model validation component. The version analyzed is ASP.NET Core 2.2 stable, released December 4, 2018.

Validations

ASP.NET Core MVC similarly supports validation as Hibernate Validator since it permits to enhance the object model with constraints using data annotation validation attributes. The referred constraints are specified at *Field Level* (*Property Level* in ASP.NET) and also at *Class Level* (*Model Level* in ASP.NET). Some of validation attributes that exist in ASP.NET

Core MVC are: *[Required], [Range], [StringLength], [CreditCard], [Editable], [EmailAddress], [Phone]*. As there may be instances where it is required to express specific validation rules, one can create custom validation attributes. This feature will be addressed next, but first its described the process with the existing/provided validation attributes. Listing 4.4 illustrates the declaration of the provided validation attributes, and it follows the same example illustrated in listing 4.1.

```
1  public class Room
2  {
3      [Key]
4      public int Id { get; set; }
5
6      public int roomNumber { get; set; }
7
8      [Range(1, 10)]
9      public int adultsCapacity { get; set; }
10
11     [Range(0, 5)]
12     public int childrenCapacity { get; set; }
13
14     [Range(0, 999.99)]
15     public decimal Price { get; set; }
16 }
```

Listing 4.4: ASP.NET Core MVC Model with Constraints Example.

After the declaration of the validation attributes, the process to verify those invariants implies that the programmer invokes a function, usually in the controller, named `IsValid`. With this, the validation on the server-side is ensured.

ASP.NET Core MVC can also enforce validation in the client-side. The programmer needs to specify two flags in a configuration file and install some dependencies, namely `jQuery`, `jQuery.Validation` and `jQuery.Unobtrusive.Validation`. The mechanism implemented can use the validation attributes and type metadata from model properties to render HTML 5 data attributes in the form elements that need validation. jQuery Unobtrusive Validation then parses the HTML 5 data attributes and passes the logic to jQuery Validate, by copying the server side validation logic to the client. The process behind the generation of web pages relies on a server-side markup language, called `Razor` [47], and the use of it is implicit when creating pages and form validations.

The default validation attributes work for most of the cases, however sometimes it is needed to specify a more detailed validation, and custom validation attributes are a great solution. It is possible to define custom validation attributes to a single property or to multiple properties (by decorating a class with it). The procedure to create custom validation attributes is relatively easy since one needs to create a new class which derives from `ValidationAttribute` and then overrides the `isValid` method. For instance, in listing 4.5 is illustrated the case if one pretends to introduce a validation that checks if a booking

object has a valid state, comparing the number of persons with the capacity available in the room.

```
public class ValidateBookingAttribute: ValidationAttribute {
    private Room _room;

    public ValidateBookingAttribute(Room room) {
      _room = room;
    }
    protected override ValidationResult IsValid(object value,
    ValidationContext validationContext) {
      Booking booking = (Booking) validationContext.ObjectInstance;

      if (booking.roomId != _room.id)
        return new ValidationResult(``Error message 1'');

      if (booking.numberOfAdults > _room.adultsCapacity ||
      booking.numberOfChildren > _room.childrenCapacity) {
        return new ValidationResult(``Error message 2'');
      }
      return ValidationResult.Success;
    }
```

Listing 4.5: Implementation of a custom validation attribute in ASP.NET Core MVC Model.

The validation of the custom validation attribute is equal to the default ones, at least in the server-side. The difficulty increases when dealing with the client-side validation of the custom validation attribute since there is no mechanism to automatically perform the same operations as it happens with the default validation attributes. The programmer is responsible for coding the necessary JavaScript validation function, and that can be somewhat difficult and time-consuming.

Analysis

The method to introduce data constraints in ASP.NET is similar to the one provided by Hibernate Validator, as it relies on annotations. The declaration of the invariants is also made directly in classes, and it ensures that it exists a unique place where the definition of constraints happen, without the need to replicate them over the server-side. The main spotlight is the ability to replicate the data conditions defined in the classes and their validation (server-side) to the client-side. With this, data validation is ensured in both the critical points of the system. The usage of the default validation attributes ensures that the effort by the programmer in the replication is minimal since it only needs some configuration steps. The main objection is related to custom validation attributes. Their existence is relevant because it allows specifying invariants that can constrain one or multiple classes, but when transposing those invariants to the client-side, the programmer needs to write to JavaScript logic to ensure those rules in the client-side. In our proposed

solution, the programmer does not need to replicate the invariants whether in the server-side or the client-side.

4.3 Ruby On Rails

Ruby on Rails is a web application framework written in the Ruby [48] programming language [25]. It is organised to conform the MVC software design pattern, and this is enforced by the combination of several packages, such as*Active Record*, *Active Model*, *Action Pack* and *Action View*, among others [25]. The most interesting package in this context is *Active Record*. *Active Record* facilitates the creation of objects whose data requires persistent storage to a database, and it is an implementation of the Active Record Pattern [25]. Martin Fowler described the Active Record Pattern [18], and it is defined as an object that carries both persistent data and behaviour which manipulates that data. The version that is analysed is Rails 5.2.2 stable, released December 4, 2018.

Validations

The Active Record package of Ruby On Rails is used as an ORM Framework, and it includes the representation of models and their data and the representation of associations between these models, among others. The validation of models before they get persisted to the database is also included.

Also as the previous frameworks, Active Record permits to enhance the model with constraints that provide common validation rules, which are called Validation Helpers. Some of these constraints are *acceptance*, *format*, *length*, *numericality*, *presence*, *uniqueness*. If these build in validation helpers does not serve the purposes, Rails also has the support to the programmer define custom validations. Those validations are addressed later in this chapter.

Each validation helper accepts an arbitrary number of attribute names, so it is possible to associate a kind of validation to several attributes. Listing 4.6 is the default approach of the declaration of the default validation helpers to each attribute or set of attributes. It includes an example of the creation and validation of an object. The validation is done using the *valid?* function call that outputs a boolean value.

```
1  class Room < ApplicationRecord
2      validates :id, :roomNumber, uniqueness: true
3      validates :adultsCapacity, :roomNumber, numericality:{ only_integer: true,
4      greater_than_or_equal_to: 1 }
5      validates :childrenCapacity, :id, numericality: { only_integer: true,
6      greater_than_or_equal_to: 0 }
7      validates :price, :numericality: { greater_than_or_equal_to: 10.0 }
8      validates :id, :roomNumber, :adultsCapacity, presence: true
9  end
10
11 Room.create(id: 0, roomNumber: 100, adultsCapacity: 2, childrenCapacity: 0,
```

```
price: 25).valid? # => true

Room.create(id: 0, roomNumber: 30, adultsCapacity: 0, childrenCapacity: 0,
price: 65).valid? # => false
```

Listing 4.6: Validation in the *Ruby on Rails* platform.

Ruby on Rails does not support client-side validations out of the box. Although there exists a gem [2] called ClientSideValidations [3] that enables programmers to include client-side validations "automatically" based on the model. Essentially, it just automates the process of generating JavaScript code that performs the validations.

If the model has validations in which the standard helpers aren't enough, the programmer can implement a custom validation strategy. It is possible to code those custom validations as a class method(*custom methods*) or as a separate class(*custom validators*). The decision is up to the programmer, although the usage of *custom validators* seems to be a better approach, in the architectural point of view, because it separates the validation layer avoiding extensive models that can affect the reliability and maintainability. Listing 4.7 illustrates the implementation of a custom validation as a separate class, that constrains the max number of children per one adult in a room.

```
class RoomMaxChildrenValidator < ActiveModel::Validator
    def validate(record)
        unless ( (self.adultsCapacity == 1 && self.childrenCapacity <= 4) ||
        self.adultsCapacity > 1 )
            record.errors[:childrenCapacity] << ``Too many children
            for one adult!''
        end
    end
end
```

Listing 4.7: Implementation of a custom validation in the *Ruby on Rails* platform.

Analysis

Ruby on Rails assembles both the ORM technique and the validation all in one. With this, it ensures the validation of the constraints when the data is persisted to the database, but also whenever the programmer explicitly implies it. The declaration of the default constraints (Validation Helpers) and the custom validations is done in a unique place, ensuring that there is no need to repeat the validation logic through multiple locations. There are several mismatches that this framework has comparing to our objectives. Ruby on Rails is a server-side framework, which does not apply entirely to our problem, as we aim for both server-side and client-side validations. Also, the custom validations are

[2] In Ruby, a gem is a library that performs a specific piece of functionality. RubyGems is the package manager for the Ruby programming language, that is used to easily manage the installation of gems, and also a server for distributing them.

[3] Extension available at: https://rubygems.org/gems/client_side_validations.

restricted to one record, meaning that it is impossible to define constraints that involve multiple models that aren't declared in the same object instance.

4.4 Mendix

Mendix is a low-code software platform that provides tools to build, test, deploy and maintain applications [36]. It offers an IDE, such as OutSystems [53], that allows defining user interfaces with building blocks and widgets, creating domain models for reading and writing data and modelling the interactions and flow control of an application. When modelling the domain model of an application, it is possible to set up data validation through the usage of validation rules [59]. The version analysed is Mendix Beta 8, released May 3, 2019.

Validations

Mendix provides a mechanism to constrain the values of the attributes of entities, called Validation Rules [59]. Those are conditions that should be satisfied before an object is committed. If the condition defined by a validation rule is not satisfied when the object is committed, the server generates a validation error, and if it occurs in the context of a form submission, it results in a message that is propagated to the user interface [59]. The properties of a validation rule are the following: *Attribute*, *Error Message* and the *Rule*. The attribute property defines the attribute to which the validation rule applies. Each validation rule can only be applied to a single attribute, invalidating the definition of rules that contains more than one attribute, and also rules between different entities. The error message is the message that is displayed to the end user whenever the condition fails to be satisfied. The rule defines which condition an attribute should satisfy, and it exists six different options, including *Required*, *Unique*, *Equals*, *Range*, *Regular expression*, *Maximum length*. Figure 4.1 illustrates the configuration of a new validation rule for a specific entity, and it is visible its structure as described above.

As said before, if the object was committed using a form, the system propagates the message to the user interface. The built-in rules work for most of the cases, however sometimes it is needed to specify a more detailed user input validation, such as the validation between different attributes, and in that case, the programmer needs to manually create the desire validation logic, as it happens in the OutSystems Platform.

Analysis

Mendix overcomes OutSystems by providing an extended implementation regarding the introduction of the concept of validations rules, in such a way that it allows to validate data before persistence with predefined conditions that are previously assigned to entity attributes. It also permits to define custom validations that occur closer to the user interface, such as the OutSystems platform. Comparing to our work, it misses the possibility

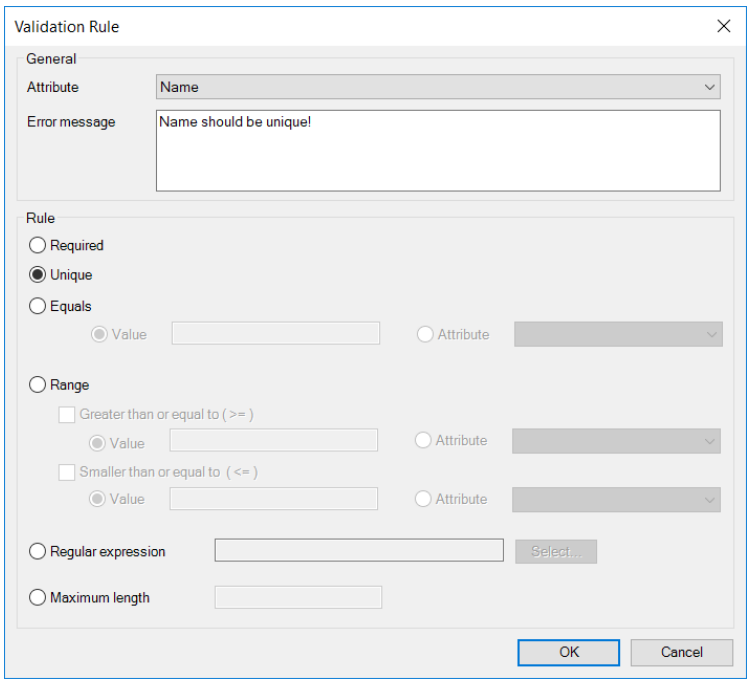


Figure 4.1: Validation Rule - Mendix.

to define validation rules that restrict two different attributes in a single instance, along with the opportunity to propagate the definition of those validation rules through the application's layers, taking into account the flow control and business logic in between.

4.5 OutSystems Forms

In 2017, Paweł Krysiak developed his master thesis [33] in collaboration with OutSystems. His work consisted of the expansion of a textual DSL, inspired by the OutSystems Visual DSL, with constructors that allow describing form validations in a declarative, easy and structured way. The main goal was to simplify the way that the validation code is added to an application. The main focus was on forms, so the mechanism was built closer to the client-side. It also aimed to remove the need for the programmer to repeat validation code in multiple layers. The DSL expansion was applied in an existing DSL developed by previous MSc students at OutSystems [6, 22, 23].

Validations

This work also used a data flow analysis technique to propagate the specification of data invariants, also referred to as validation rules. These were defined directly in the Input Widgets of a Form in a Web Screen. They can also be expressed as a new datatype and used as the expected type for an input. The concept of validation rules is similar to our own since the same properties constitute them.

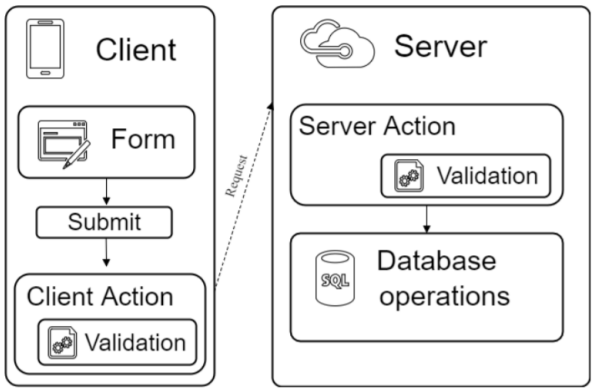


Figure 4.2: System architecture - Paweł Krysiak Dissertation [33].

Analysis

Paweł's work was the first to approach the concept of data validation in the OutSystems platform. It contributed with a study that will be taken into consideration for possible future implementations to include expressing data invariants and generating validation code automatically. Our work aims to complement it by going in the opposite direction, to introducing data invariants directly in the data layer, namely in entities, and providing an analysis that propagates and generates validation code associated to them. Our work also aims to fulfil future work left by Paweł, mainly the definition of invariants in entities, and the specification of a data flow analysis *path sensitive* mechanism.

4.6 Summary

All of the frameworks previously introduced provide the capability to specify data constraints. They generally implement some default constraints, that can be used in most of the common situations, such as for limiting the range of values for a numeric attribute, or ensure that an attribute is not null or to validate if a specific text respects the email address format. They also provide mechanisms that permit to express custom constraints, so that more precise data validation can be applied. The propagation of the validations to the client-side exists only in some of the frameworks. These ensure that it happens automatically or by the use of external libraries that enforce that automation. In the server-side, the frameworks define that it is up to the programmer to explicitly invoke the previously specified validation logic. Some of the frameworks ensure the automatic validation of the constraints when the data is persisted to the database. Mendix platform [36] introduces the possibility to specify data invariants directly in entities, ensuring that these are validated before data persistence. However, it misses the opportunity to propagate the invariants to different layers of an application. Finally, Paweł's dissertation [33] fulfils many requisites that our work also pretends to accomplish. It allows defining data invariants, in web pages, and propagating them throughout the application's layers. I also

include the automatic generation of validation code. However, it lacks the specification of invariants in entities, and it does not take into consideration the conditional structures implicit in the program's flow.

It was identified and briefly analysed two more tools, not listed in this document: The framework Grails [24] and a DSL named WebDSL [63, 64]. Grails is a web application framework, and its validation mechanism is equal (with the same principals and features) to the one provided by the Active Record package of Ruby On Rails, that is described in this chapter. WebDSL is a DSL for modelling web applications with rich data models. It was not described because it did not bring any relevant characteristic, comparing to the ones already identified and characterised.

Table 4.1 presents a quick comparison of all the related work identified and the features that our work fulfils. Table 4.2 shows the direct comparison between Paweł work and ours.

Mechanisms / Tools / Frameworks	Default Constraints and Custom Constraints	Automatic Validation before Persistence	Server-Side / Client-Side Validations	Automatic Analysis and Suggestion of Validations	Generation of Validation Code
Hibernate Validator	☑	☐	☐	☐	☐
Hibernate Validator + Hibernate ORM	☑	☑	☐	☐	☐
ASP.NET Core MVC	☑	☐	☑	☐	☐
Ruby On Rails	☑	☑	☐	☐	☐
Mendix (Validation Rules)	☑	☑	☐	☐	☑
OutSystems Forms (Pawel's work)	☑	☐	☑	☑	☑
Our Solution	☑	☐	☑	☑	☑

Table 4.1: Comparison of related work and our solution.

Work Comparison	OutSystems Forms (Pawel's work)	**Our Solution**
Data Flow Analysis	*Path Insensitive*	*Path Sensitive*
Constraints specified in	Web Forms	Entities
Prototype	DSL	On *Service Studio* code

Table 4.2: Comparison of Paweł work and our solution.

Chapter 5

Technical Approach

In this chapter, we will present the technical aspects of our solution. We describe in detail our data flow analysis technique, and changes made to the OutSystems language model that allows to specify data invariants. For a more detailed explanation of the prototype implementation, please refer to Chapter 6.

5.1 General Overview

The use of static analysis techniques allows one to obtain information regarding useful properties of analysed programs, and within this context, it enables us to conclude which validation rules should be applied and validated in the different layers of the application. The static analysis technique used in this work is the data flow analysis applied to the program control flow graph (CFG) [30, 56]. Our data flow analysis is a *backward analysis*, and its sensitivities are *flow-sensitive* and *path-sensitive*. In our work, the definition and reasoning of the logic needed to reach the necessary conclusions are achieved through a Datalog specification, which uses the Prolog engine called SWI-Prolog [57]. This engine and its usage are described in detail under section 6.1. We decided to do the data flow analysis more declaratively versus a more imperative approach, such as implementing iterative fixed-point algorithms [42]. The reason behind such decision relies on the fact that since it is declarative, it results in a faster and more agile implementation through the import of libraries, which allow for an interpretation of a Datalog program without loss of efficiency during its execution. Also, it is easier to define the inference/propagation rules and to see an immediate result.

Figure 5.1 presents an overview of our mechanism. We start by analysing the OutSystems application's model, including the data conditions specified in entities and the application's data flow. From that analysis, we extract facts that are used as an input for

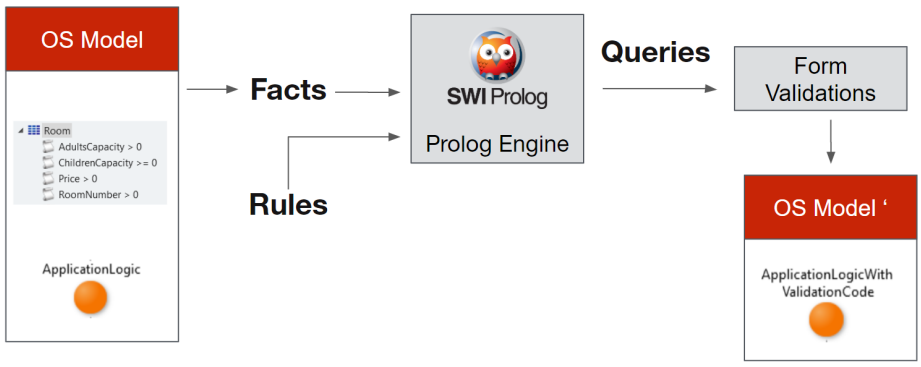


Figure 5.1: Mechanism Overview.

the Prolog engine. We also give the Datalog rules as an input, that allow concluding new facts. Then, after the processing of the Prolog logic, we perform queries, whose result is the data validations that should occur in a specific program point. Finally, we use that set of validations to generate validation code, which is included in the previous application logic and resulting in a new application's model.

In the remaining sections of this chapter, we will describe our static analysis technique and the changes made to the OutSystems language model. Then, in the next chapter, we describe the implementation of the prototype based on the statements of those sections.

5.2 Analysis for Validation Rules

In this section, we will present more details about our data flow analysis technique, including the description of the Datalog facts and rules. These Datalog elements allow to manipulate and propagate the expressions of the data invariants, or validation rules, as well to collect the expressions of conditional structures imposed by the programmer.

The collection of conditional expressions are essential because it allows us to conclude conditional validation rules. Throughout the program's flow, starting from the nodes that manipulate entities, to the program points in which validations needed to be inserted, all possible paths are analysed. It may exist simple flows, where data is persisted without any modification or validation, or there may be more complex ones, with changes to the data and conditional structures. To deal with all these different flows, we defined new facts that made the analysis *path-sensitive*. We describe these facts and all others throughout this chapter.

In the following subsection, we will introduce our Datalog facts and rules, along with examples to enrich the explanations. These examples come as a result of the action flow analysis illustrated on the left of the figure 5.2. The action contains labels (illustrated by a rectangle) identifying each node, which we use later in the examples. As we purpose generating validation code in all system's layers, we need to add an "virtual "node" to the CFG. It symbolises the client-side of applications, namely the web pages that trigger the actions'execution. We illustrate it in the top left corner of the figure. The action's flow is

simple, and it has two conditional structures with assignments that express an example of a business rule. On the right of this figure, we illustrate an entity with 4 data invariants, also referred to as validation rules (below the entity name). We do the detailed definition of validation rules in Chapter 5.3. Meanwhile, the label of each of these validation rules corresponds to the invariants' expressions.

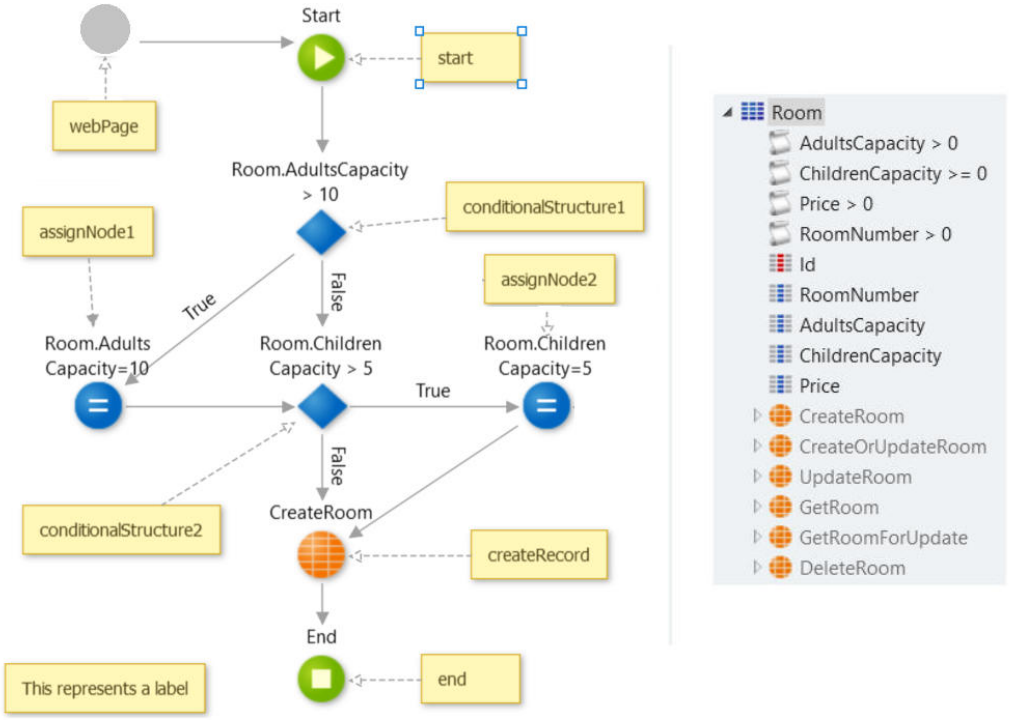


Figure 5.2: Example of an OutSystems action, used as a CFG, (on the left) and an Entity with Validations Rules (on the right).

5.2.1 Data Flow Analysis

Datalog enables the querying of a set of facts without having to specify in detail how to achieve an answer. Being Datalog a logic query language, we have introduced our Datalog rules, in a logical form, as *inference rules*: if all premises are true (facts above the line), then all conclusions must be true (facts below the line). The rest of this section presents both the facts directly extracted from the CFG and the rules, based on which the existing facts generate even more facts. That contribute towards finding a solution to a given query.

To formally define the proposed static analysis, the following notations are introduced. We denote by $\mathcal{N}$ the set of all action nodes of an application, such that $n, n1, n2 \in \mathcal{N}$; we express by $\mathcal{EN}$ the set of all entities that belong to the application being analysed, with $en \in \mathcal{EN}$; we denote by $\mathcal{R}$ the set of the existing validation rules in all entities, with $r \in \mathcal{R}$; we express the set of imposed conditions by the programmer by $\mathcal{C}$, such that $c \in \mathcal{C}$; we denote by $\mathcal{V}$ the set of local variables in actions, with $\mathcal{V}$; we express by the set of all

entity attributes, such that $a \in \mathcal{A}$; finally, we denote by $\mathcal{E}$ the set of expressions used in assigns, with $e \in \mathcal{E}$.

5.2.1.1 Facts

Given a CFG, we first need to extract facts that constitute the knowledge base of our Datalog program. In the OutSystems platform context, those facts are extracted directly from the application's model, since their logic is already a CFG, as described in section 2.2.2. The Datalog facts can be divided into three groups: the ones related to rules, program's flow and nodes. We introduce the following fact related to rules:

rule(en, r)	States that entity *en* has the rule *r*.

We now introduce the following facts related to the program's flow:

follows(n1, n2)	States that node *n1* can be followed by node *n2*.
followsIF(n1, n2, c)	States that node *n1* is a conditional node, which is followed by node *n2*, and that *c* is the imposed condition at node *n1*.

Finally, we present the following facts associated with nodes:

assign(n, v, e)	States that in node *n* the expression value *e* is assigned to var. *v*.
input(n, en, a, e)	States that in node *n* the expression value *e* is used as the argument for attribute *a* of entity *en*.
node(n)	States that node *n* does not make any effect in the analysis.

We now present brief examples of the generation of each fact previously introduced. To do so, let's apply the fact generation to the graph in figure 5.2. For simplicity, we follow the order of the facts presented earlier. The *rule* fact arises from the analysis of the validation rules defined in the entity manipulated by the *entity action*. We illustrate in listing 5.1, the four *rule* facts produced for this example. In this fact, we save the rule condition and the entity in which it is defined.

```
1 rule(Room, Rule1: AdultsCapacity > 0).
2 rule(Room, Rule2: ChildrenCapacity >= 0).
3 rule(Room, Rule3: Price > 0).
4 rule(Room, Rule4: RoomNumber > 0).
```

Listing 5.1: Example *Rule* facts.

The *follows* fact arises from the flow analysis, and it is meant to identify unconditional paths in the graph. In the example, we can notice many paths of this type, for instance, the path from the *CreateRoom* node to the *End* node.

```
follows(webPage, start).
follows(start, conditionalStructure1).
follows(assignNode1, conditionalStructure2).
follows(assignNode2, createRecord).
follows(createRecord, end).
```

Listing 5.2: Example *Follow* facts.

The *followsIf* fact is quite similar to the previous one, and it is used when there are conditional structures (If) in the program's flow. For each conditional structure, we save the True and False path of its expression. Listing 5.3 presents these facts that were generated in the example.

```
followsIf(conditionalStructure1, assignNode1, (Room.AdultsCapacity > 10)).
followsIf(conditionalStructure1, conditionalStructure2, not(Room.AdultsCapacity > 10)).
followsIf(conditionalStructure2, assignNode2, (Room.ChildrenCapacity > 5)).
followsIf(conditionalStructure2, createRecord, not(Room.ChildrenCapacity > 5)).
```

Listing 5.3: Example *FollowsIf* facts.

The *assign* fact is used when an assignment is made. In the example, the assignments made were always attributions of values to variables, but there exist several different types of assignments., such as assignments between local variables or between variables of different layers, among others. Listing 5.4 presents these facts that were generated in the example.

```
assign(assignNode1, Room.AdultsCapacity, 10).
assign(assignNode2, Room.ChildrenCapacity, 5).
```

Listing 5.4: Example *Assign* facts.

The *fact* input is used when there is an *entity action* node, and it maps the entity's attributes to the local variable that contains the data to be persisted. Listing 5.5 presents these facts that were generated in the example.

```
input(createRecord, Room, RoomNumber, RoomForm.Room.RoomNumber).
input(createRecord, Room, AdultsCapacity,RoomForm.Room.AdultsCapacity).
input(createRecord, Room, ChildrenCapacity, RoomForm.Room.ChildrenCapacity).
input(createRecord, Room, Price, RoomForm.Room.Price).
```

Listing 5.5: Example *Input* facts.

Finally, the *node* fact is generated when there is a node that does not modify the expressions of validation rules, meaning that it only propagates the rule itself. Listing 5.7 presents these facts that were generated in the example.

```
node(webpage). node(start).
node(conditionalStructure1). node(conditionalStructure2). node(end).
```

Listing 5.6: Example *Node* facts.

Appendix A presents a listing with all the generated facts for the example of figure 5.2. In the next section, we describe the Datalog rules that allow to derive even more facts and answer queries.

5.2.1.2 Rules

The declaration of a fact in implies that a statement is always true. Rules can be viewed as an extension of a fact, in the sense that it is a fact with added conditions that have also to be satisfied for it to be true. Therefore, based on the referred facts, we have defined a set of Datalog rules to express the properties that allow for propagating the data conditions, defined in the entities, and the restrictions imposed by the programmers, defined in the application logic layer, through the several layers that constitute an application. We have specified the following Datalog rules:

hasRuleIn(n, c, r) At the **entry** of node n, there is a rule r that needs to be validated under condition c.

hasRuleOut(n, c, r) At the **exit** of node n, there is a rule r that needs to be validated under condition c.

As stated before, in section 5.2.1, we defined the Datalog rules as *inference rules*, and Figure 5.3 illustrates the specification of these.

$$\frac{rule(en,r) \quad \overline{input(n,en,a,e)}}{hasRuleIn(n,True,r[\overline{e/a}])} \ (1)$$

$$\frac{follows(n1,n2) \quad hasRuleIn(n2,c,r)}{hasRuleOut(n1,c,r)} \ (2) \qquad \frac{followsIF(n1,n2,c1) \quad hasRuleIn(n2,c2,r)}{hasRuleOut(n1,c2 \wedge c1,r)} \ (3)$$

$$\frac{assign(n,v,e) \quad hasRuleOut(n,c,r)}{hasRuleIn(n,c,r[e/v])} \ (4) \qquad \frac{node(n) \quad hasRuleOut(n,c,r)}{hasRuleIn(n,c,r)} \ (5)$$

Figure 5.3: *Inference Rules.*

We now explain the meaning of each *inference rule*:

Inference Rule (1) - Express that at the entrance of a specific node, all the validation rules are described by the *rule* and *input* facts. The premise *input* has a line over it symbolising a set of Datalog facts. The set comprises all *input* facts that contains the same entity as the *rule* fact, in their *terms* (section 3.5.2 describes *terms*). The conclusion states that, **each** occurrence of a in r is replaced by the expression e.

Inference Rule (2) - The inf. rules (2) and (3) are associated with the program's control flow. This specific inf. rule is applied when there exists a flow between two nodes without conditional structures.

Inference Rule (3) - It applies to flows with conditional structures. It is important to notice that this inf. rule makes our approach *path sensitive*: including condition c as a premise for the rule captures information about the path that was followed.

Inference Rule (4) - It is associated with the attribution of values to variables, corresponding to Hoare's assignment axiom [26]. Recurring to the assignment axiom, we state that, the occurrence of v in r is replaced by the expression e.

Inference Rule (5) - Applies to the nodes that do not affect the manipulation of the expressions, since they only propagate the validation rules, which are equal at the node's entry and exit.

To enrich the description given above, we will present the application of three inference rules and their conclusions, namely inf. rules (1), (3) and (4), following the action flow presented in figure 5.4.

The analysis starts from an *entity action* node, and the first inference rule used is the inf. rule (1). The assumptions necessary to get the conclusion of inf. rule (1) are the following:

```
1  // Facts required:
2  rule(Room, Rule1: AdultsCapacity > 0).
3  rule(Room, Rule2: ChildrenCapacity >= 0).
4  rule(Room, Rule3: Price > 0).
5  rule(Room, Rule4: RoomNumber > 0).
6
7  input(createRecord, Room, RoomNumber, RoomForm.Room.RoomNumber).
8  input(createRecord, Room, AdultsCapacity,RoomForm.Room.AdultsCapacity).
9  input(createRecord, Room, ChildrenCapacity, RoomForm.Room.ChildrenCapacity).
10 input(createRecord, Room, Price, RoomForm.Room.Price).
```

Listing 5.7: Example *Node* facts.

```
1  // Conclusion obtained:
2  hasRuleIn(createRecord, True, (Rule2: ChildrenCapacity >= 0)
3      [RoomForm.Room.RoomNumber/RoomNumber]
       [RoomForm.Room.AdultsCapacity/AdultsCapacity]
       [RoomForm.Room.ChildrenCapacity/ChildrenCapacity]
       [RoomForm.Room.Price/Price]
4  )
5  // The simplified version:
6  hasRuleIn(createRecord, True, (Rule2: RoomForm.Room.ChildrenCapacity >= 0))
```

Listing 5.8: Inference Rule (1) Example.

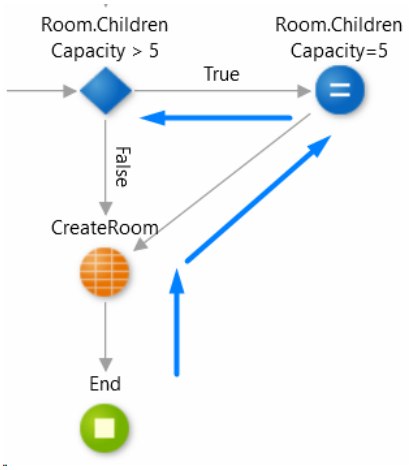


Figure 5.4: Action Flow Example.

Given these facts, which correspond to the assumptions, the conclusion of inf. rule (1) for the validation rule *Rule2: ChildrenCapacity*$> = 0$ is illustrated in listing 5.8.

Having obtained the conclusion of inf. rule (1), illustrated earlier, we will present the application of the inf. rule (4). It is important to note that between the *entity action* node and the *Assign* node there is an unconditional path where inf. rule (2) is applied, but we do not present its demonstration as it is quite trivial. Applying inf. rule (4) on the *Assign* node returns the conclusion illustrated in listing 5.9.

```
// Facts required:
assign(assignNode2, Room. childrenCapacity, 5).
hasRuleOut(createRecord, True, (Rule2: RoomForm.Room.ChildrenCapacity >= 0)).

// Conclusion obtained:
hasRuleIn(assignNode2, True, (Rule2: RoomForm.Room.ChildrenCapacity >= 0)
    [5/RoomForm.Room.ChildrenCapacity]
).

// The simplified version:
hasRuleIn(assignNode2, True, (Rule2: 5 >= 0)).
```

Listing 5.9: Inference Rule (4) Example.

Finally, listing 5.10 presents the application of the inf. rule (3) and its conclusion.

```
// Facts required:
followsIf(conditionalStructure2, assignNode2, (RoomForm.Room.ChildrenCapacity > 5)).
hasRuleIn(assignNode2, True, (Rule2: 5 >= 0)).

// Conclusion obtained:
hasRuleOut(conditionalStructure2,
    (RoomForm.Room.ChildrenCapacity > 5) ∧ True, (Rule2: 5 >= 0)
```

Listing 5.10: Inference Rule (3) Example.

5.2.1.3 Queries

Given the definition of the Datalog rules, and after populating the knowledge base of the Datalog program, one can make a query to find values to substitute into the query variables such that it is satisfied. We do not specify any particular predicate for queries; instead, we use the predicate *hasRuleIn(n, r)*. We provide the specific node *n* corresponding to the program's location for the propose of getting the set of validation rules *r*.

5.3 OutSystems Model

One of the purposes of the previously described data flow analysis is the propagation of the validation rules defined in the data model. This requires expanding the data layer implementation to enable one to specify validation rules. This section provides a more detailed description of the OutSystems model and validation rules. We will present both the current model and the extended one with validation rules.

In this dissertation, and for the purpose of simplicity, we have described a subset of the OutSystems language consisting of elements that are important to our static analysis, such as *Screens*, *Actions* and *Entities*. Some of these elements had already been introduced in chapter 2, however we are now presenting a meta-model which enriches said definitions. Listing 5.11 illustrates a subset of the language's meta-model presented in Appendix B. We will describe the listing structure. The declared classes corresponds to OutSystems language elements. We are specifying OutSystems *Action*'s elements, *Web Screens* and *Entities*. For each class we declare their *Properties* and *Children*. *Properties* symbolise attributes, and *Children* represent aggregation relations. For instance, class *Screen* has, or is made up of, a *Name* property, a set of *Widgets* and a set of *ScreenActions* (lines 3,4 and 5 in Listing 5.11). The meta model's meta model is presented in [35].

In the next section, we define validation rules, and explain the approaches to introduce them in the model presented earlier. Finally, we present the new extended model.

5.3.1 Invariants/Validation Rules

Given all definitions and specifications of the static analysis, described in the previous section, we needed to change the previously presented model, so that it allows the programmer to express invariants/validation rules on the application. We firstly describe validation rules, followed by the approaches made for the model expansion, and finally the present the new extended model.

```
<MetaModel>
    <Class name="Screen">
        <Property name="Name" type="Text" />
        <Child name="Widgets" type="Widget" />
        <Child name="ScreenActions" type="Action" />
    </Class>

    <Class name="Action">
        <Property name="Name" type="Text" />
        <Child name="Nodes" type="ActionNode" />
    </Class>

    <Class name="ActionNode">
        <Property name="Target" type ="ActionNode" />
    </Class>

    <Class name="Start" base="ActionNode" />
    <Class name="End" base="ActionNode" />

    <Class name="Assign" base="ActionNode">
        <Child name="Assignments" type="Assignment" />
    </Class>

    <Class name="If">
        <Property name="TrueLink" type ="ActionNode" />
        <Property name="FalseLink" type ="ActionNode" />
        <Property name="Condition" type="Expression" />
    </Class>

    <Class name="Execute" base="ActionNode">
        <Property name="Action" type="Action" />
        <Child name="Arguments" type="Argument" />
    </Class>

    <Class name="Entity">
        <Property name="Name" type="Text" />
        <Child name="Attributes" type="Attribute" />
        <Child name="EntityActions" type="EntityAction" />
    </Class>

    <Class name="Attribute">
        <Property name="Name" type="Text" />
        <Property name="DataType" type="Type" />
    </Class>
</MetaModel>
```

Listing 5.11: Subset of the latest version of OutSystems Meta-model (before our changes).

The specification of a logical assertion is not sufficient for our purposes, so, we have introduced the concept of validation rules. Validation rules are composed by the following properties: *Name*, *Condition* and *Error Message*. The name property helps to identify, visually on the platform, the validation rule. The condition is a boolean expression that specifies the intended invariant. It may include one attribute or several different attributes. The error message is a text that is displayed if the evaluated condition reveals to be false.

For the prototype development, it was necessary to enrich the OutSystems model with the notion of validation rules. In a first approach, the validation rules were defined at the attribute level, and it was intended to ensure that each rule would be associated with only one attribute of the entity. This approach would make it possible to directly associate error messages with input widgets (see chapter 2.2.1) that exists in the form of a page. Thus, since each input widget is associated with an attribute of an entity, the error message would be displayed next to the widget that invalidated the validation rule condition. However, this approach was discarded, as it was limited. The reason is that it would not be possible to define rules that used more than one attribute in the condition, therefore making it impossible to have a richer rule set.

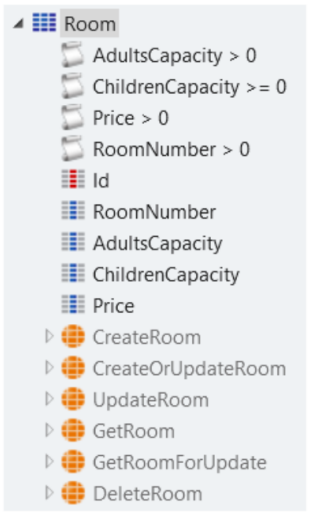


Figure 5.5: Extended Entity View.

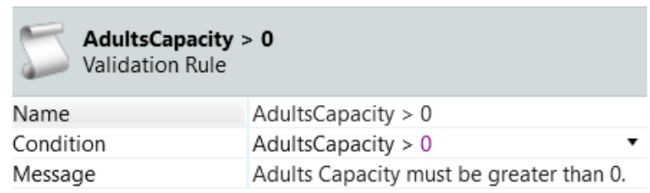


Figure 5.6: Validation Rule View.

Hence, in a second approach, the specification of validation rules has moved to the entity level, allowing the creation of rules that in their condition includes more than one attribute. For instance, and remembering the running example, it is now possible to specify a validation rule that ensures that, when registering a new reservation, the check-out date is always after the check-in date. This is an example of its added value. Listing 5.12 presents the changes to the OutSystems meta-model, which now includes the validation rule specification. Figure 5.5 illustrates visually an entity in OutSystems, that already encapsulates the definition of validation rules. That entity has four validation rules. Figure 5.6 shows, in more detail, the definition of the first validation rule.

```
1    <!-- Previous elements remain unchanged (except Entity) -->
2    ...
3
4    <!-- Entity Modification (new child) -->
5    <Class name="Entity">
6        <Property name="Name" type="Text" />
7        <Child name="Attributes" type="Attribute" />
8        <Child name="EntityActions" type="Action" />
9        <Child name="ValidationRules" type="ValidationRule" />
10   </Class>
11
12   <!-- New class -->
13   <Class name="ValidationRule" >
14       <Property name="Name" type="Text" />
15       <Property name="Condition" type="Expression" />
16       <Property name="ErrorMessage" type="Text" />
17   </Class>
18 </MetaModel>
```

Listing 5.12: OutSystems Meta-model changes.

This chapter described our technical approach, including the data flow analysis technique, and the changes made to the OutSystems model. In the next chapter, we will describe the implementation of our prototype, including the generation of facts and the generation of the validation code.

Chapter 6

Implementation

This chapter describes the implementation of the prototype of our mechanism. Our prototype was implemented in Service Studio to define validation rules in *Entities*, which are propagated to *Screen Actions* and validated by the generated code. The remaining of this chapter describes the prototype implementation, including libraries used and explanations on how we processed to generated the Datalog facts and the validation code.

6.1 Data Flow Analysis

The data flow analysis is a critical feature of our work and prototype. It allows for analysing the control flow graph of the application being evaluated, with the main objective of propagating the validation rules that have been previously defined in the data layer. It collects these validations rules and conditional statements that exist in the application's logic layer, thus producing a set of conditional validation rules that must be validated in the program's specific point for which the analysis was meant. As stated before, in chapter 5.2 and section 6.2.2, the analysis consists of a Datalog program that uses a Prolog interpreter, with a table-oriented resolution method (*SLG resolution*), as its resolution engine. In the previous section, we introduced the model syntax in which the static analysis was being applied.

The remainder of this section details how the generation of facts occurs, and how the generation of the validation code happens after obtaining an answer from the resolution engine.

Language Syntax

The prototype implementation with the proposed mechanism was developed directly on the Service Studio module. As described earlier in section 5.3.1, the changes in the language only consisted of adding the concept of validation rules. The OutSystems language is quite broad and complex, but in the context of this dissertation, only a small subset of the language was used. Appendix B contains a simplified version of the OutSystems language meta-model, which helps understand the following presented procedures.

6.1.1 Extraction of Facts

We generate facts by examining the application model. Abstractly, it is required to generate the program's CFG (see section 3.5), as this is where the data flow analysis is usually performed. Following this step, one can start the actual static analysis and generation of Datalog facts. As described in section 2.2.2, an OutSystems action consists of a set of nodes that congregate into an oriented graph, meaning that these actions can be already interpreted as CFG, which simplified our prototype implementation. Each node in an OutSystems action, or action node, has a set of properties that defines it as an object, and for each type of fact we intend to generate, we need to look at particular properties, which provide us with enough information to define the actual Datalog facts.

Before introducing and describing each fact and the properties needed to specify it, we describe the type of graph analysis, including the algorithm, that we implemented into our prototype. As in our static analysis, we analyse the most recent version of the CFG, there was no need to create another structure for saving it, given that we only needed to transverse the CFG and extract relevant information. The generation of facts occurred at the same time as the transverse, in order to avoid unwanted extra time complexities, as would occur if the generation occurs after the transversal.

We perform a Depth First Search (DFS) in the graph, using a recursive function with a structure to record the nodes already visited in order to avoid processing a node more than once and, thus, duplicating the Datalog facts. We store these nodes in a Dictionary data structure (for the O(1) lookup time complexity), using the node's unique identifier as its key. It should be noted that, in our work, we have not processed loop mechanisms, meaning that if the action contains such type of mechanisms, they do not modify any validation rule specification. As it is pointed out later, we settled the analysis of loop nodes for future work.

We will now briefly describe the logic behind the generation of each Datalog fact, as introduced in section 5.2.1.1, divided into the same three groups as before: facts related to rules, program's flow and nodes.

6.1.1.1 Rules Facts

In this group, we only defined one fact (*rule* fact) that is addressed to the validation rule objects that are defined in the entities of an application. Each fact represent a single object, and it specifies the association between the rule and the entity where it was annexed to. Each rule object is composed of a name, boolean expression and error message.

***Rule* fact** We create a new rule fact after all the other fact groups are created by evaluating the entities exists in the application's data layer. For each entity, we obtain the set of validation rules, and then for each, we declare a new *rule* fact, as shown in Algorithm 1.

Algorithm 1 Creation of the *rule* facts

```
1: procedure NewEntityRuleFact(entity)                    ▷ Given an entity
2:     for all validationRule in entity.ValidationRules do
3:         Create a new fact rule(entity.ID, validationRule.ID, validationRule.Condition)
4:     end for
5: end procedure
```

6.1.1.2 Program's flow Facts

In this group, we have defined two facts that represent the application's control flow. One of the facts addresses the flows without conditional statements (*follows* fact), and the other (*followsIF* fact), by the opposite, collects the conditional statements and both the true and false flows.

***Follows* fact** The *follows* fact is used to state that exists a path from a node n to its target node $n1$, being n a non-conditional node. Algorithm 2 is used several times in distinct procedures through our implementation, and it just contains an API call to declare the fact.

Algorithm 2 Creation of the *follows* facts

```
1: procedure NewFollowsFact(node, targetNode)
2:     Create a new fact follows(node.ID, targetNode.ID)
3: end procedure
```

***FollowsIF* fact** The *followsIF* fact states that exists a path from a node n to its target node $n1$, being n a conditional node. We call the procedure *GenerateFactsWithinIfNode* illustrated in Algorithm 3, whenever the graph traversal finds an If node.

Algorithm 3 Creation of the *followsIF* facts

```
1: procedure GenerateFactsWithinIfNode(node)
2:                                          ▷ First, it deals with the the true branch
3:     nextNode ← GenerateDatalogFacts(node.TrueLink.TargetNode)
4:     NewIfFollowsFact(node, nextNode, true)
5:                                          ▷ Now, it deals with the the false branch
6:     nextNode ← GenerateDatalogFacts(node.FalseLink.TargetNode)
7:     NewIfFollowsFact(node, nextNode, false)
8:     NewNodeRuleFact(node)                ▷ See Algorithm 6
9: end procedure

10: procedure NewIfFollowsFact(node, nextNode, isTrueLink)
11:     if (isTrueLink) then
12:         Create a new fact followsIF(node.ID, nextNode.ID, node.Condition)
13:     else
14:         falseCondition ← not(node.Condition)
15:         Create a new fact followsIF(node.ID, nextNode.ID, falseCondition)
16:     end if
17: end procedure
```

6.1.1.3 Nodes Facts

In this group, we have defined three facts that represent the nodes. One of the facts is used to specify assignments, another addresses the entity actions nodes, and the last one represents nodes that do not modify the conditions of the validation rules.

***Assign* fact** The *assign* fact states that in a specific node exists an assignment to a specific variable. These facts are particularly important since they contribute to the mapping that exists in the modification of the conditions in the validation rules. So, when the analysis comes across with an assign node, the conditions expressions likely change. Algorithm 4 shows the procedures implemented to declare *assign* facts.

***Input* fact** The *input* facts, as the previous ones, are essential to the static analysis. These facts are used to state that an entity action is called in the program's flow, and they map the variables used in the actions to the entities and their attributes in the data layer. The procedures in Algorithm 5 shows how the *input* facts are defined.

***Node* fact** The *node* fact, as stated before, represent nodes that do not modify the specification of the validation rules, meaning that these nodes only propagate the validation rules without any effect. For more context, section 5.2.1.2 explains the inference rules and the use of the *node* facts. The procedure to define this fact is simple, Algorithm 6 describes it.

Algorithm 4 Creation of the *assign* facts

```
1: procedure GenerateFactsWithinAssignNode(previousNode, assignNode)
2:     for all assignment in assignNode.Assignments do
3:         if (assignment is simple) then
4:                                          ▷ Deals with simple variables and values
5:             Processing of the variable and value
6:             variable ← processedVariable     ▷ Deals with simple variables and values
7:             value ← processedValue
8:         else
9:                                          ▷ Deals with compound variables and values
10:            Processing of compound variable and compound value
11:            variable ← processedVariable
12:            value ← processedValue
13:        end if
14:        NewAssignFact(assignNode, variable, value)
15:    end for
16:    NewFollowsFact(previousNode, assignNode)                ▷ See Algorithm 2
17: end procedure

18: procedure NewAssignFact(node, variable, value)
19:     Create a new fact assign(node.ID, variable, value)
20: end procedure
```

Algorithm 5 Creation of the *input* facts

```
1: procedure GenerateFactsWithinEntityActionNode(node)
2:     argument ← node.Argument
3:     entity ← argument.TargetEntity
4:     for all attribute in entity.Attributes do
5:         Processing of the argument to match the entity attribute
6:         argValue ← processedArgument
7:         NewInputFact(node, entity, attribute.Name, argValue)
8:     end for
9: end procedure

10: procedure NewInputFact(node, entity, attributeName, argValue)
11:     Create a new fact input(node.ID, entity.ID, attributeName, argValue)
12: end procedure
```

Algorithm 6 Creation of the *node* facts

```
1: procedure NewNodeRuleFact(node)
2:     Create a new fact node(node.ID)
3: end procedure
```

After enumerating the several procedures for each type of node analysed, we disclose the algorithm responsible for examining the graph of a given web screen action, illustrated in Algorithm 7. It analysis the existing program's flow until all the nodes of the graph are visited. According to the type of nodes found during traversal, the procedures already presented are invoked.

Algorithm 7 DFS to transverse the action's graph and to specify Datalog facts

```
1:  procedure GraphTraversal(webScreenAction)
2:      visitedNodes ← []
3:      node ← webScreenAction.getFirstNode()          ▷ Obtain the Start Node of
    webScreenAction
4:      webPageNode ← webScreenAction.Parent   ▷ Define a new node to represent the
    Web Screen
5:      GenerateDatalogFacts(node, visitedNodes)
6:      NewNodeRuleFact(webPageNode)
7:      NewFollowsFact(webPageNode, node)
8:  end procedure

9:  procedure GenerateDatalogFacts(node, visitedNodes)
10:     if (visitedNodes.ContainsKey(node.Id)) then
11:         return node
12:     else
13:         visitedNodes.Add(node.ID, node)
14:     end if
15:     if (node.Type is of type Nodes.Start or node.Type is of type Nodes.End) then
16:         NewNodeRuleFact(node)
17:     end if
18:     if (node.Type is of type Nodes.EntityAction) then
19:         GenerateFactsWithinEntityNode(node)
20:     end if
21:     if (node.Links.Count == 0) then
22:         return node
23:     end if
24:     if (node.Type is of type Nodes.If) then
25:         GenerateFactsWithinIfNode(node)
26:     else
27:         for all link in node.Links do
28:             nextNode ← link.TargetNode
29:             if (node.Type is of type Nodes.Assign) then
30:                 GenerateFactsWithinAssignNode(node, nextNode)
31:             else
32:                 NewFollowsFact(node, nextNode)
33:             end if
34:         end for
35:     end if
36:     return node
37: end procedure
```

In the next section, we present an example of the analysis and its results. Low-code platforms such as the OutSystems platform simplify the analysis process as there is an abstraction layer that makes it easier to analyse the control flow. Nevertheless, as stated before, our static analysis method can be applied to any language or application; however, a more considerable implementation effort is required.

6.1.2 Outputs of the Analysis

In this section, we use the example given in Figure 5.2, and present the application of the static analysis and its results. As we already illustrated how we generate the facts and the rules usage, we now present the set of facts that were generated during the graph traversal, and the set of validation rules that must applied as soon as data arrives to the server.

```
1  rule(Room, Rule1: AdultsCapacity > 0).
2  rule(Room, Rule2: ChildrenCapacity >= 0).
3  rule(Room, Rule3: Price > 0).
4  rule(Room, Rule4: RoomNumber > 0).
5
6  follows(webPage, start).
7  follows(start, conditionalStructure1).
8  follows(assignNode1, conditionalStructure2).
9  follows(assignNode2, createRecord).
10 follows(createRecord, end).
11 followsIf(conditionalStructure1, assignNode1, (RoomForm.Room.AdultsCapacity > 10)).
12 followsIf(conditionalStructure1, conditionalStructure2, not(RoomForm.Room.AdultsCapacity
       ↪ > 10)).
13 followsIf(conditionalStructure2, assignNode2, (RoomForm.Room.ChildrenCapacity > 5)).
14 followsIf(conditionalStructure2, createRecord, not(RoomForm.Room.ChildrenCapacity > 5)).
15
16 assign(assignNode1, RoomForm.Room.AdultsCapacity, 10).
17 assign(assignNode2, RoomForm.Room.ChildrenCapacity, 5).
18
19 input(createRecord, Room, RoomNumber, RoomForm.Room.RoomNumber).
20 input(createRecord, Room, AdultsCapacity,RoomForm.Room.AdultsCapacity).
21 input(createRecord, Room, ChildrenCapacity, RoomForm.Room.ChildrenCapacity).
22 input(createRecord, Room, Price, RoomForm.Room.Price).
23
24 node(webpage).
25 node(start).
26 node(conditionalStructure1).
27 node(conditionalStructure2).
28 node(end).
```

Listing 6.1: Example *Rule* facts.

Listing 6.1 presents all the facts generated. Given that list, one queries the engine on the validation rules that should be verified in the beginning of the action, ensuring data validation as soon as possible on the server-side. The purpose of submitting a query is

to find values to substitute into the variables such that the query is satisfied. To obtain the validation rules that should be applied at the action beginning, the following query is submitted, using the *hasRuleOut(n, r)* rule (see section 5.2.1.2):

```
1 // R - variable
2 ?- hasRuleOut(webPage, R).
```

The *webPage* corresponds to the node where we pretend to discover the set of validation rules (set of R). Listing 6.2 presents a subset of the query results, that corresponds to the path in which all conditions of conditional structures are true. The remaining query results can be seen in Appendix C.

```
1 // * = hasRuleOut (for simplicity)
2
3 *(webPage, (RoomForm.Room.AdultsCapacity > 10 ∧ RoomForm.Room.ChildrenCapacity > 5),
      ↪ Rule1: 10 > 0)
4 *(webPage, (RoomForm.Room.AdultsCapacity > 10 ∧ RoomForm.Room.ChildrenCapacity > 5),
      ↪ Rule2: 5 >= 0)
5 *(webPage, (RoomForm.Room.AdultsCapacity > 10 ∧ RoomForm.Room.ChildrenCapacity > 5),
      ↪ Rule3: RoomForm.Room.Price > 0)
6 *(webPage, (RoomForm.Room.AdultsCapacity > 10 ∧ RoomForm.Room.ChildrenCapacity > 5),
      ↪ Rule4: RoomForm.Room.RoomNumber > 0)
```

Listing 6.2: Solutions obtained from the static analysis.

The results subset reveals the validation rules that must be tested in the *webPage* node. The expressions of validation rules are already mapped according to the local scope of the *Web Screen*, meaning that it considers the names of local variables. The expressions obtained are conditional, meaning that they will be tested when certain flow conditions are established. For instance, the validation of the derived expressions only occurs when both the values of *AdultsCapacity* and *ChildrenCapacity* are greater than ten and five, respectively. Sometimes, and according to the logic imposed by the programmers, it may surge trivial proofs, where the expressions of validation rules do not depend on values from introduced by the user in the form. For instance, notice the line 3 of Listing 4, where, if the flow condition is verified, the expression to be tested will always be satisfied, and its proof is trivial.

6.1.3 Generation of Validation Code

Given the answers obtained in the previous section, we will now present how we proceeded to generate the code that validates the validation rules. As mentioned earlier, our prototype aims to have validation in the application logic layer immediately as soon as data arrives. The decision to insert validations in the logic layer was based on a strategic decision. At an early stage, the learning of the OutSystems language model focused on Web applications, leaving out Mobile apps. Therefore, the mechanism prototype has

always been focused on Web applications and is only available for this type. Due to time constraints, we had to postpone the implementation in Mobile apps, but at the current stage, it is easy to achieve. This point, however, immediately invalidates the insertion of validation on the client-side. In mobile applications, there exist specific actions (*Client Actions*) that enable the execution of logic in the user's device. In Web applications, such an option does not exist, at least in the current version of the OutSystems platform. Since we intended an implementation without intervention on the compiler, the insertion of validations moved to the server-side. The current version of our prototype generates logic in *Web Screen Actions*, which is called right after the Start Node and before the logic created by the programmer. We will now describe in detail the generation of the validation code.

Up to this point, the analysis and its results were invisible to the programmer. We will now describe how it becomes visible within the platform, namely through the generation of validation code. Given an error-free implementation of code generation, it is now possible to visually conclude that static analysis is well specified and achieves its purpose. The validation code generation is triggered by clicking the first option in the *Web Screen action*'s context menu in Service Studio. It firstly triggers the data flow analysis, and then the code generation. Currently, the process is manual, but it is not the most desired behaviour. Due to time constraints, we had to postpone the automatic call of this procedure. However, we identified some situations where it should happen: creation or modification of a validation rule; entity modifications (e.g., adding or removing attributes); changes of the actions' logic; among others. In our prototype, it is possible to see and modify the generated code, which is also not the most desired behaviour. The generated code should be invisible to the programmer as it does not introduce any added value. However, it would have to be indicated to the programmer that such validation code exists. But as it involves user interface planning, it has not been studied or discussed in much detail and was left as future work.

The previous section explained how we obtain the set of conditional validation rules for the running example. For simplicity, we now explain how the code is generated based on a more straightforward use-case.

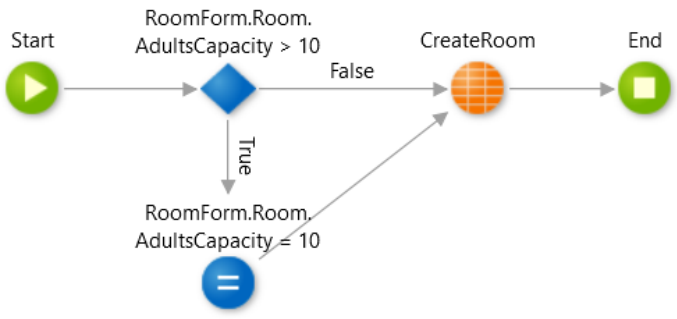


Figure 6.1: Small action used for illustrating the code generation (BEFORE).

Figure 6.1 illustrates an action with a simple flow, in which there is a conditional structure and an assignment, along with a call of a *Server Action*, and more specifically an *Entity Action*, which persists the data in the database. For simplicity, we consider that there exists only one validation rule in the Room entity, namely the Rule1 (*Rule1: AdultsCapacity* > *0*). According to this action and validation rule, the validation code that is required will validate the boolean condition of that same rule, and its application is conditional.

The validation code generation relies on the analysis of the conclusions obtained by the data flow analysis. Accordingly to these conclusions, we proceed to the generation of the code that validates the set of validation rules, whether conditional or not, in a specific point of the program. As already mentioned, that location is the beginning of the action called by submitting data from a Web screen. The generated code consists of the use of *If* and *Assign* nodes, as well as a particular node responsible for triggering errors. The particular node makes it possible for the error messages defined in the specification of a validation rule to appear on the Web page.

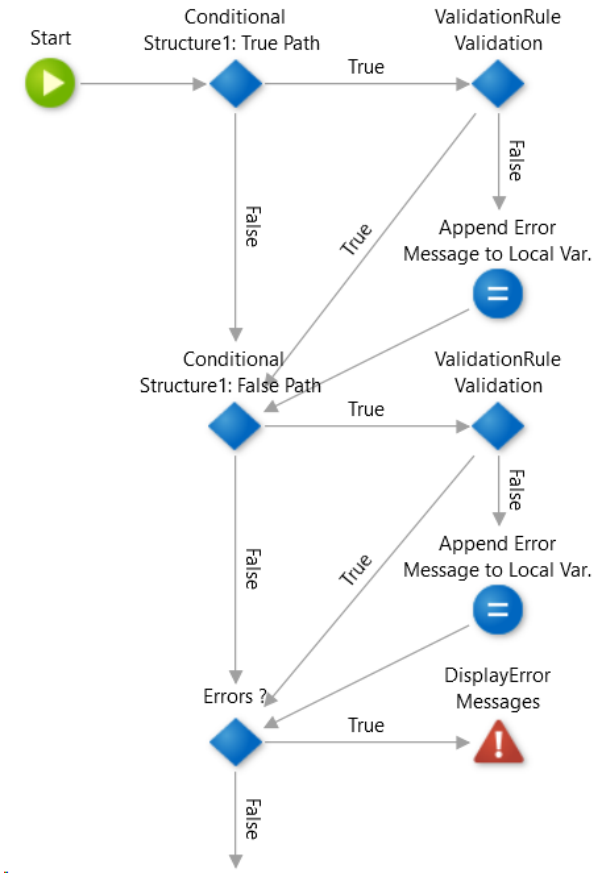


Figure 6.2: Overview of the generated validation code.

Figure 6.2 illustrates an overview of the generated validation code structure. At the bottom of the illustration, it is visible two nodes that are always generated, namely the *If* node labelled ”Errors“ and the *Exception* node with the label ”DisplayError Message“.

These two nodes validate, at runtime, if there were any violations of the validation rules, and if so, an error is thrown on the Web page. The error text contains the error messages of the violated validation rules. The remaining of the generated code is subject to the number of conditional structures and validation rules.

We will now analyse the flow of the logic of figure 6.2, starting at the *Start* node. The first node after *Start* represents a boolean conjunction of one of the possible paths from the action's beginning to the node that handles the entity. According to the conditional structures found, the validation rules are increasingly conditionals on the different paths. If there is only one condition structure in the action's flow, two *If* nodes are generated, as illustrated in figure 6.2. These nodes express the *True Path* and the *False Path* of the condition structure. If there is not any condition structure in the action's flow, then we only generate an *If* node with the condition expression *True*. However, if it exists more than one conditional structure in the action's flow, then our data flow analysis technique combines the various expressions of these structures as a boolean conjunction. In those circumstances, the code generation algorithm creates only one *If* node for each boolean expression conjunction. It also groups the various validation rules that are evaluated under that conjunction as an optimisation. For each validation rule, it is created two different nodes, an *If* node and an *Assign* node. The *If* node validates the propagated expression of the validation rule. Whenever the runtime validation of the expression fails, the error message of the validation rule is appended to a local variable, using the *Assign* node. This local variable stores all the error messages, and it is used in the exception node already discussed. If the submitted data does not violate any of the expressions, the programs' flows continue to the logic's coded by the programmers. Otherwise, as described, an exception is thrown, ending with the flow of the actions.

Figure 6.3 illustrates the action, represented in figure 6.1, after generating the validation code, following the details given above.

6.2 Libraries Overview

In previous dissertations [22, 33] developed at OutSystems, which used static analysis techniques, the prototype has always been developed using an existing textual language that represents a subset of the OutSystems visual DSL. These prototypes used *NodeJS* for the server-side and *AngularJS* for the client-side and also used the tools Xtext and Eclipse. For the static analysis, they used a library called IRIS-Reasoner that provides a Datalog interpreter and an API which allows for creating the Datalog program through objects. In our work, we have established that the prototype would be developed directly on the OutSystems Platform. This decision led to the search and later use of a library that integrates with .NET (C#). In the remainder of this section, we will briefly describe two libraries that were analysed and used for implementing the data flow analysis. Only one of them is being used in the final version of the prototype, named SWI-Prolog.

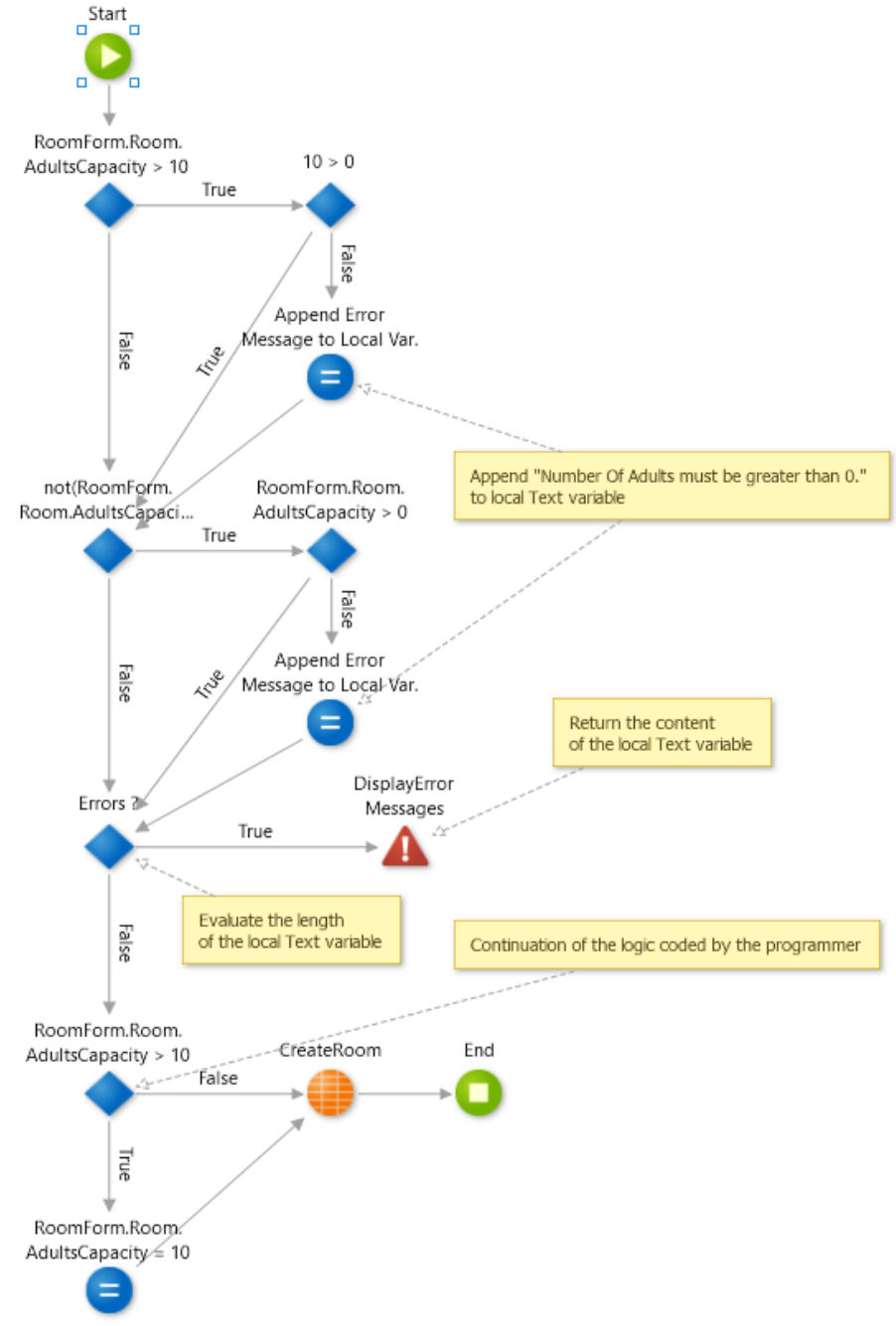


Figure 6.3: Small action used for illustrating the code generation (AFTER).

6.2.1 Z3 Theorem Prover

Z3 is a Satisfiability Modulo Theories (SMT) solver from Microsoft Research that combines several theory solvers into a combined framework. SMT solvers can quickly solve complex constraints over Boolean, integer and first-order logic predicates, as well as various data structures, such as lists, arrays or bit-vectors. SMTs are NP-Complete and decidable, meaning that the solver always gives an answer, which is an important point when dealing with complex problems.

We used a built-in extension of Z3 called muZ, which is an engine for fixed points with logical constraints. The default fixed-point engine is a bottom-up Datalog engine. The Z3 library provides an API that allows for defining relationships/predicates, including rules and facts, as well as asking questions and getting answers. However, some limitations have caused its use not to be part of the final version of the prototype. One of the restrictions is related to the Datalog engine of this library, where it does not support infinite domains. This limitation means that it is only possible to define relations over Booleans, Bit-Vectors or Finite Domain Values. Initially, we defined the relations, which include both the rules and facts, using a Bit-Vector approach. This implies the use of mapping between bit-vector values and variables or values in C#, which is an unwanted approach for our requirements. However, for an initial validation, its use was sufficient. We later found another limitation when iterating the inference rules to add more complexity and cover more use cases. Following the statements described in chapters 5.2 and 5.3, there emerged the need to collect the flow restrictions enforced by programmers and to map the variables in the validation rules. These two requirements led to the modification of the inference rules, so that it was possible to conclude under which conditions the validation of each validation rule should occur. In addition to this, the mapping of the condition variables would have to be done on-the-fly to obtain results mapped to the correct scope. Thus, the use of bit-vectors would eventually become a bottleneck. The use of Z3 as a Datalog program resolution engine has been set aside. Its use could, however, still be useful for simplifying expressions, but due to time constraints, we had to postpone it for future work.

6.2.2 SWI-Prolog

The use of the Z3 library was quite limited to the needs that our solution required. Therefore, it was necessary to find a different library that met the same initial requirements. In our final approach, the definition and reasoning of the logic, required to reach the necessary conclusions, has been achieved achieved through a Datalog specification, which uses the Prolog engine called SWI-Prolog [57]. SWI-Prolog was chosen primarily for its integration with .Net (C#), and for supporting the table-oriented resolution method called *SLG resolution*. This method ensures that a response is obtained even in highly chained predicate sets, by storing intermediate answers in memory [9, 58]. The resolution method commonly used by Prolog (*SLD resolution*) does not ensure the completion of questions

and often recomputes intermediate results [9]. The use of a Datalog interpreter would be preferential, considering all of its benefits, but we couldn't find a library to integrate with .Net (C#).

Despite this detail, the use of this interpreter proven itself sufficient as regards defining, reasoning and concluding the conditions under which validation rules should be validated at a given point in the system. The interaction with SWI-Prolog, as with Z3, is done using an API which enables the definition of rules, facts and questions. However, this library allows for a broader range of data types that could be used for these statements. It mainly allows using *Strings* as a data type, which was a significant improvement over Z3. The use of *strings* brought some benefits that contributed towards this library being used in the final version of the prototype. Firstly, the visualisation of the rule, fact and question definitions became much clearer, given that no intermediate mapping was necessary. This also made it easier to detect any errors in the implementation or use of the API, compared to the previous alternative. Secondly, the conditions of validation rules are Boolean expressions that, ultimately, can be interpreted as a sequence of characters forming a String. As such, the manipulation of conditions throughout the static analysis is now done on-the-fly - that is to say, it is embedded in the inference rules, which greatly simplifies the implementation process.

The SWI-Prolog version used in the prototype is SWI-Prolog 8.0.3-1, and the API for C# to SWI-Prolog is SwiPICs 1.1.60605.0 [1]. The use of these versions introduces a limitation on the architecture of the operating system on which Service Studio is installed, since its use was only possible in the 64-bit version.

6.3 Limitations

Currently, our mechanism has some limitations, either related to the static analysis specification or prototype implementation. The limitations of the prototype are partly associated with the boundaries of the static analysis specification. Firstly, we do not analyse repetition structures. The analysis of such structures is important because they certainly produce effects on the propagated expressions of the validation rules. We examine and consider conditional structures, namely *If* blocks, to obtain an analysis *path sensitive*and to allow concluding conditional validation rules. However, we do not analyse *Switch* blocks, being left as future work.

Validation rules are specified in entities, allowing the expression of a given condition to consider various attributes of that same entity. However, the possibility of being able to create validation rules that could use attributes from different entities would be a huge asset. This would make it possible to define much more complex invariants. However, its usefulness may be more constrained to the data layer, as in other applications' layers there may not be possible to obtain all the data needed to validate the invariant.

[1] https://github.com/SWI-Prolog/contrib-swiplcs

Currently, the prototype is only available to Web Applications. However, we do not foresee any difficulties extending it to Mobile Applications in the future. The only application entry point used in our analysis is the Web Screen element with a *Form* widget. The OutSystems model has other application entry points, such as REST endpoints. We decided to limit the prototype to deal only with Forms since it is the most common way of data transmission from users. The analysis of the action's flows is limited to a subset of the OutSystems language. We consider the following model elements: *Start*, *End*, *If*, *Assign*, *Sub Action Calls*, *Entity Actions*, *Local Variables and Input/Output Parameters*. As stated before, we do not analyse *Switch* and *ForEach* nodes. Even so, there exists other nodes are important and commonly used, such as *Aggregates/SQL nodes* (Allows to fetch and compute data from the database). These allow to fetch and compute data from the database, producing a list of results, that are used by programmers throughout the action's flow in many different ways. Due to time constraints, we left them as future work.

Chapter 7

Evaluation

In this chapter, we will describe the evaluation of our mechanism. The analysis was made on a large system developed in OutSystems, where we propose to conclude and evaluate several points, related to the development in OutSystems, as well as the adaptability and functionality of our engine and prototype.

7.1 Analysis Overview

The evaluation of this work consisted in the study of a system developed in OutSystems and the extraction of information that allowed us to answer several questions we considered crucial. These questions are associated with development in OutSystems and the feasibility and adaptability of our mechanism, including its benefits. We analysed only one system largely due to time constraints. The analysis of more systems or applications would be beneficial as it would help to enrich the inferred conclusions. However, the system analysed is vast (described in section 7.2), and was developed by several teams, so there is a level of variability in the analysis supporting its evaluation.

The possibility of performing usability tests was analysed; however such tests were omitted for two reasons. Firstly, our implementation never considered a visual interface to users as an essential requisite. The implication is that many modifications could be made concerning the user interface. For instance, the user can view and change the generated validation code. However, it should be invisible to the user and only change automatically based on existing validation rules and action's coded logic. Secondly, there were practical constraints such as time and a lack of consensus regarding the ideal requirements to produce a high-quality usability test.

The remainder of this section defines the analysis goals and explains the method applied.

7.1.1 Objectives

In this subsection, we describe the objectives of the analysis conducted. These final points attempt to respond to questions that we consider crucial to evaluate the studied system as well as our mechanism and prototype.

The generic analysis objectives questions are as follows:

- How is the system structured?
- How is the data validation organised?
- Where does data validation exist?
- Is the validation code replicated in the different system modules?
- Is the validation code replicated in the different system layers?

Based on those listed questions, we can analyse data validation, code replication and other aspects of low-code platform development. In addition to the previous results, the analysis intends to prepare and present conclusions concerning the integration of our mechanism in the system. For this, we will answer the following set of questions:

- Which are the most common data invariants?
- How is validation done?
- What is the prototype's coverage level?
  - Under what circumstances is it not possible to analyse the data flow?
  - Under what circumstances does the analysis not produce the expected validation code?
- Would the mechanism bring benefits?

Section 7.3 presents the analysis conclusions derived from these questions. In the next subsection, we explain the analysis method applied.

7.1.2 Method

The analysis method was comprised of a manual code verification of the various OutSystems modules that form the system under study. The ownership of the system is vested in one of OutSystems customers, and there has not been any specific provision of documentation associated with it. The lack of formal documentation leads to increased time effort to analyse the context and structure of the various modules.

We will now describe the method of analysis for obtaining the key information that allowing us to answer the interrogations noted. The initial step was a quantitative analysis of the modules, entities and Web screens. After the characterisation of these elements, we performed a qualitative analysis. This analysis begins by classifying the modules in terms of service, namely, user interface, business logic and data layer. Given this classification, we then started a manual verification process of each module belonging to the user interface to locate Web screens with forms. For each Web screen found under

these conditions, the applications' flow was analysed in detail. The flows found were mostly of high complexity. They contain several conditional and repetition structures, as well as the invocation of functions and actions defined in different modules. Once we have identified a data flow from a Web screen to an entity, we then collected information for further analysis and cross-checking. For instance, we gathered information about layers where exists data validation and if it exists manual replication between them. We also extracted, through a more detailed manual analysis, data invariants interpreted from the flow. With these invariants, we evaluated the coverage of our prototype, by checking the invariants possible to be specified in our mechanism.

7.2 Evaluated System

We analysed the various topics mentioned before on a large system, probably one of the largest systems ever built using OutSystems. It consists of around 700 applications modules, where there are around 2000 entities and 2500 Web Screens. All the modules of this system are over 500MB in size. This system was also used in the evaluation of a previous dissertation [52].

After an initial analysis, it was concluded that the implementation structure, in terms of OutSystems modules, follows a division according to a *Three-Tier* architecture. That said, we have identified the following group of modules:

- **UI** - modules that constitute the user interface;
- **BL** - modules that contain the business logic;
- **CS** - modules that aggregate the database entities.

There are still other groups of modules, but their identification was omitted as they are not relevant to our work. From the entire set of modules, there are 89 UI modules, 316 BL modules and 126 CS modules. It is important to notice that, for each UI module, there may exist dozens of Web pages. The same reasoning logic applies to other types of modules, as for the number of server actions and entities. It is a system of considerable size and complexity.

Describing the characteristics identified in each of these groups of modules, the user interface (UI) modules represent the graphical interface of the system, comprising several Web screens according to their expected function in the system. As mentioned, it is a system of considerable size and complexity, so we only analysed Web screens that transmit data to the server, specifically through forms, with the intent of data persistence. The business logic (BL) modules represent the layer dedicated to the rules of how the business operates. This layer lies between the presentation layer (UI modules) and the data layer (CS modules). These modules are mostly composed of server actions that process the data coming from UI modules according to the system business logic. CS modules compose the data layer, where it is defined the system entities, data structures and server actions.

We can distinguish two unique categories of server actions in these modules. The first category consists of relatively simple actions geared towards data retrieval. The other category includes actions that act as an interface for invoking *Entity Actions*, and they usually have logic that validates the already existence of a record in the database.

In the next subsection, we will characterise the analysed sample of the referred system.

7.2.1 Sample Characterisation

In our evaluation, we discerned two different samples. The first one is a larger sample, which was evaluated in less detail. It is the knowledge base used to answer the general group of questions referred to in section 7.1.1. The second sample is smaller and underwent more detailed analysis. It was used to evaluate our mechanism and prototype.

The first sample comprised of 4 UI modules, which were considered the most important to the system context. Within this set of modules, we randomly selected 50 Web screens, of which only 20 where Web pages with submission forms, for analysis. This set of Web screens manipulates close to 80 different entities. The remaining Web Screens were mostly record listing screens and templates for PDF documents. For the second sample, it was necessary to reduce the number of UI modules under analysis, and consequently, the number of Web Screens and entities. The set was reduced to 2 UI modules, and we analysed 4 different Web Screens that handled 10 entities.

7.3 Results

In this section, we will discuss the results obtained from the analysis. We divide this section into (1) global and (2) specific results. First, we describe the global results obtained associated with the generic analysis objectives. Then, we disclose the specific conclusions corresponding to the flexibility and functionality analysis of our implementation.

(1) Global Results

The analysed system has an excellent division of the layers that constitute an application, through the modularity used. It was possible to identify the layers (the system's *three-their* architecture) even without previously knowing the system or its documentation.

In this system, we detected two approaches to validating data. The first and most consistent approach consists of data validation, through manually defined code, only at the presentation layer (UI modules). We identified 17 Web Screens, where the data flow followed this validation approach. As this system is a Web application, where the logic of *Screen Actions* runs on the server-side, validation only occurring in the server is not problematic. However, if it was a mobile app, the data could eventually arrive corrupted to the server, as the validation using only screens actions would be insufficient. The next approach validates data in the presentation and logic layers, and it was detected in the data flow of only 3 Web screens. We conducted a more careful analysis to identify

a pattern for the implementation of the approach. For instance, one of the possible patterns that we analysed was whether different Web screens invoked that server Action, and whether in the same module. In the end, we concluded that the validation code in the two layers was the same, finding that it was a manual replication.

We identified the replication of the validation code for a set of attributes of an entity in two Web Screens of different modules. The finding indicates that there are no actions that centralise the validation code, thus removing code replication, and allowing it to be invoked by various modules and screen actions.

In this sample, we did not find validation of data from the forms in the data layer (CS modules). As stated, the most common logic patterns consist of either directly invoking the entity action that persists the data without validation, or checking whether a given record already exists in the database.

To conclude, we argue that data validation should take place at another essential point in the data flow, namely in or near the database. By validating the consistency of invariants immediately before their persistence, it ensures that no data corruption has occurred and it also proves the accuracy of the existing coded logic.

(2) Specific Results

In this section, we will present the conclusions obtained in the detailed analysis of the system. We attempt to answer questions related to our proposed mechanism, as well as questions related to the integration of the prototype with the system under analysis.

As described in section 7.2.1, we performed our detailed analysis on a small set of Web screens. For each, we analysed the data flow between the various layers. We set the collection of those Web screens based on data flow complexity, amount of code intended for data validations, and the number of modified entities. These criteria are important as they allowed the selection of data flows and invariants susceptible to be tested with our mechanism, especially with the implementation of our prototype. We describe these tests later in this section.

The analysed Web pages are forms with variable complexity that turn into flows with varying complexity, and that manipulate a larger or smaller number of entities. In the analysed sample, we only found data validations in the presentation layer, namely in the ”Save“ Web screen actions. The ”Save“ actions correspond to the screen actions that ultimately persist the information by having a flow than ends in an entity action.

The validation logic found is common to all Web screens, and it consists of *If* and *Assign* nodes. Each *If* node validates a specific data invariant, and each *Assign* node defines a boolean variable and an error message. After examining the *If* nodes, we identified different types of data invariants. The most common type is the validation of only one attribute of a given entity. This type of data invariant is simple, and our prototype does not impose any restrictions on its specification. Additionally, two more variations of invariants were identified in much smaller quantities. One of these types corresponds to the validation

of invariants that restrains various attributes of different entities. In contrast, the other type validates invariants that constrain attributes of the same entity. Our work allows the definition of data invariants that constrain attributes of a given entity. However, it is not possible to express data invariants that depend on a set of attributes of different entities. In the current version, the data flow analysis technique and the prototype associate each validation rule with a single entity (as discussed in section 6.3).

The set of Web screen actions from this sample were too large to test our prototype. As a result, we narrowed the list of screen actions to two. These two "Save" actions correspond to the actions with the highest and lowest complexity of the sample. Although it is a somewhat limited set in quantity, it proved to be relevant for concluding on essential aspects of our prototype and work. We will now describe these two actions' flows:

Most complex logic This logic contains over 380 OutSystems nodes, which are counted using the sum of nodes present in all actions invoked since the Screen Action "Save" of a given Web Screen. Of this set of nodes, the "Save" action uses 234 nodes, whereas 57 corresponds to data validations. In this particular action, the set of validation nodes verifies invariants of 11 different entities. Most invariants are simple. They validate a single attribute of an entity. However, some exceptions were found. Four *If* nodes expressed invariants that used attributes from different entities. Also, 5 *If* nodes exposed invariants that constrained multiple attributes of the same entity. Visually, the code block responsible for data validation consumed excessive space of the action logic, adversely affecting the overall code readability.

Least complex logic This logic contains 24 OutSystems nodes, which were counted using the same reasoning as for the most complex logic. The "Save" action defines 13 nodes, with 6 of these being used for data validation. The action persists only in one entity. We inferred this after analysing all program paths that arise from action flow. The gathered data invariants are simple, and they validate the emptiness values of attributes.

Following this brief description of both logics, we summarise the integration of our mechanism into this system. As mentioned earlier, we tested the prototype only in the two Web screen actions described. The data invariants gathered consisted of a single attribute restriction, referred to as a simple invariant, and multiple attribute constraint within only one entity, referred to as a compound invariant. For the most complex logic, we extracted 4 simple and 1 compound invariants. We defined these invariants in two different entities. For the less complex logic, we deduced 3 simple invariants, corresponding to all data validations present in the screen action.

Using the prototype in the most complex action was proved feasible, but it required modifications to its flow. One of the modifications is related to us failing to considering repetition structures, namely *ForEach*, and the *switch* control structure. To avoid unexpected errors during the facts extraction and the generation of the validation code,

we removed these blocks of code. Another modification concerns nodes that were not considered in the implementation, such as nodes with built-in functions for manipulation of lists or nodes that validate security policies. We also needed to remove them to avoid unforeseen exceptions. By changing the OutSystems logic of the more complex logic data flow adhering to the previous modifications, it was possible to propagate the expressions of the validation rules. Regarding the application of the mechanism to the less complicated flow, it was possible to propagate the validation rules without any problem, as the flow does not contain repetition structures or nodes not contemplated in the implementation.

This analysis allowed us to conclude that the mechanism developed in this dissertation fulfils most of the requirements found in the evaluated system. Due to our mechanism and implementation allowing the specification of simple and compound entity invariants, it covers a significant portion of the set of invariants detected in the system. Although only one system has been evaluated, we assume that this type of invariants will be more common in applications. The possibility of defining invariants between multiple entities would be a feature that would help to cover the remaining invariants. This is considered necessary and has been left for future work. The propagation of the validation rules expressions requires changes in the action's logic in the current version. This procedure should not be necessary for any applications or logic, and it is due to limitations in our technique and prototype implementation.

Chapter 8

Concluding remarks

The main contribution of this dissertation is a static analysis technique that, given a set of data invariants specified in the data layer, analyses the program's flows and propagates them throughout the several layers that constitute an application. Having a set of data invariants in a specific program point, it is then possible to generate automatically validation code associated with them. By propagating the specification of data invariants, or validation rules, across the various layers of a system, it is possible to introduce data validation and, ultimately, ensure data integrity. Our analysis is sensible to execution paths, also referred to as *path sensitive*, meaning that it collects all the flow restrictions imposed by the programmers, resulting in the conditional validation of data invariants.

Additionally, we developed a proof of concept implemented in the OutSystems platform, that uses our data flow analysis and generates validation code. The code generated ensures that exists data validation in the server whenever it exists a form submission, removing the programmer's obligation of writing that code and providing faster feedback to users. As we prototyped directly on the OutSystems platform source code, all the work that has been done can be directly extended and completed.

Our technique was evaluated using a large system, probably one of the largest systems ever built using OutSystems. This analysis allowed us to conclude aspects considered as fundamental. The analysis concluded that the system only performs data validation in the presentation layer, assuming that the remaining flow does not invalidate any of the data invariants. We argue that data validation should take place in all the system's layers, especially in or near the database. By validating the data before its persistence, it ensures corruption-free data, and it also proves the correctness of the coded logic. We also concluded that our mechanism and implementation lacks some features to be fully functional on a complex system, as the one evaluated. The most important one is the non-consideration of repetition structures, which are immensely used in applications. To

be able to test the prototype, it was necessary to remove this and other types of nodes from the coded logic. We also did not give much attention to the user interface, which led to no having done usability tests to prove that the mechanism increases the programmer's performance.

8.1 Future Work

Despite that the main objective of this work, creating an invariant propagation mechanism that aims to ensure data integrity, was successfully achieved, it can still be improved. There are two main points for future work to be done. The first one is to improve our analysis limitations; the second point is to complete and polish our prototype to be used and tested by any OutSystems user.
Our invariant propagation mechanism has the following future work to be done:

- **Conditional structures analysis** : Not all conditional structures are being considered for our analysis. *If* blocks are used but *Switch* blocks are not;
- **Iterative structures analysis** : We do not analysed any type of iterative blocks, which may change the expressions of the validation rules;
- **Database Validations** : Although we defined the introduction of database validations as a goal, we did not manage to achieve it, due to time constraints. Performing further checks automatically near to the database would provide an additional layer of security to an application, making it even more robust;
- **Prototype Interaction** : We did not consider the visual aspect presented on the platform as a critical requirement. Therefore, some changes can be made. For instance, the generated code is visible and can be edited by the programmer. As stated before, this is not the most desired behaviour;
- **Usability Tests and others** : Performing usability tests, among others, such as performance tests, would allow us to get more consistent conclusions of our work.

Bibliography

[1] F. E. Allen. "Control flow analysis." In: *Proceedings of a symposium on Compiler optimization* -. Vol. 5. 7. New York, New York, USA: ACM Press, 1970, pp. 1–19. ISBN: 1555-5275. DOI: 10.1145/800028.808479.

[2] *ASP.NET Documentation | Microsoft Docs*. URL: https://docs.microsoft.com/en-us/aspnet/.

[3] *Bean Validation - Bean Validation 2.0 (JSR 380)*. URL: https://beanvalidation.org/2.0/.

[4] J. Blackburn, G. Scudder, and L. Van Wassenhove. "Improving speed and productivity of software development." In: *IEEE Transactions on Software Engineering* 22.12 (1996), pp. 875–885. ISSN: 00985589. DOI: 10.1109/32.553636.

[5] L. Cardelli. "Type systems." In: *ACM Computing Surveys* 28.1 (Mar. 1996), pp. 263–264. DOI: 10.1145/234313.234418.

[6] L. Carvalho, J. Costa Seco, and H. Lourenço. *The View Update Problem in the OutSystems Aggregate Language (MSc thesis)*. NOVA University of Lisbon, 2016.

[7] S. Ceri, G. Gottlob, and L. Tanca. "What You Always Wanted to Know About Datalog (And Never Dared to Ask)." In: *IEEE Transactions on Knowledge and Data Engineering* (1989). ISSN: 10414347. DOI: 10.1109/69.43410.

[8] S. Ceri, G. Gottlob, and L. Tanca. *Logic Programming and Databases*. Springer-Verlag Berlin, Heidelberg ©1990, 1990. ISBN: 0-387-51728-6.

[9] W. Chen and D. S. Warren. "Tabled evaluation with delaying for general logic programs." In: *Journal of the ACM* 43 (1996), pp. 20–74. ISSN: 00045411. DOI: 10.1145/227595.227597.

[10] E. Codd. *The Relational Model for Database Management : Version 2*. Addison-Wesley, 1990, p. 538. ISBN: 0201141922. URL: https://dl.acm.org/citation.cfm?id=77708.

[11] M. Das, S. Lerner, and M. Seigle. "ESP: path-sensitive program verification in polynomial time." In: *Proceedings of the ACM SIGPLAN 2002 Conference on Programming language design and implementation - PLDI '02*. New York, New York, USA: ACM Press, 2002, p. 57. ISBN: 1581134630. DOI: 10.1145/512529.512538.

[12] *Documentation - 5.4 - Hibernate ORM*. URL: http://hibernate.org/orm/documentation/5.4/.

[13] *Documentation - Hibernate Validator*. URL: http://hibernate.org/validator/documentation/.

[14] *Eiffel*. URL: https://www.eiffel.org/.

[15] *Extend Logic with Your Own Code - OutSystems*. URL: https://success.outsystems.com/Documentation/11/Extensibility_and_Integration/Extend_Logic_with_Your_Own_Code.

[16] *Extensions - OutSystems*. URL: https://success.outsystems.com/Documentation/11/Extensibility_and_Integration/Extend_Logic_with_Your_Own_Code/Extensions.

[17] L. D. Fosdick and L. J. Osterweil. "Data Flow Analysis in Software Reliability." In: *ACM Computing Surveys* 8.3 (1976), pp. 305–330. ISSN: 03600300. DOI: 10.1145/356674.356676.

[18] M. Fowler. *Patterns of enterprise application architecture*. 1st. Addison-Wesley Longman Publishing Co., Inc. Boston, MA, USA, 2003, p. 533. ISBN: 0321127420. URL: https://dl.acm.org/citation.cfm?id=579257.

[19] T. Freeman and F. Pfenning. "Refinement types for ML." In: *ACM SIGPLAN Notices* 26.6 (June 1991), pp. 268–277. ISSN: 03621340. DOI: 10.1145/113446.113468.

[20] H. Garcia-Molina, J. D. Ullman, and J. Widom. *Database Systems: The Complete Book 2*. 2nd ed. Prentice Hall Press Upper Saddle River, NJ, USA, 2008, p. 1248. ISBN: 9780131873254. URL: https://dl.acm.org/citation.cfm?id=1450931.

[21] C. Garion and L. van der Torre. "Design by Contract Deontic Design Language for Multiagent Systems." In: *Proceedings of the 2005 international conference on Agents, Norms and Institutions for Regulated Multi-Agent Systems*. Springer-Verlag, 2006, pp. 170–182. ISBN: 3-540-35173-6, 978-3-540-35173-3. DOI: 10/bkm439.

[22] S. Gonçaves, J. Costa Seco, and H. Lourenço. *Otimização automática de aplicações web usando templates client-side (MSc thesis)*. NOVA University of Lisbon, 2014.

[23] B. Grácio and J. Costa Seco. *Agregado: Compilar Sistemas NoSQL na Plataforma OutSystems (MSc thesis)*. NOVA University of Lisbon, 2015.

[24] *Grails Framework*. URL: https://grails.org/.

[25] D. H. Hansson. *Ruby on Rails*. URL: https://rubyonrails.org/.

[26] C. A. R. Hoare. "An axiomatic basis for computer programming." In: *Communications of the ACM* 12.10 (Oct. 1969), pp. 576–580. ISSN: 00010782. DOI: 10.1145/363235.363259.

[27] K. Huizing and R. Kuiper. "Verification of Object Oriented Programs Using Class Invariants." In: *Proceedings of the Third Internationsl Conference on Fundamental Approaches to Software Engineering: Held as Part of the European Joint Conferences on the Theory and Practice of Software, ETAPS 2000* (2000), pp. 208–221. URL: https://dl.acm.org/citation.cfm?id=651261.

[28] A. Hunt and D. Thomas. *The pragmatic programmer : from journeyman to master*. 1st. Addison-Wesley Longman Publishing Co., Inc. Boston, MA, USA, 2000, p. 321. ISBN: 020161622X. URL: https://dl.acm.org/citation.cfm?id=320326.

[29] *Java Software | Oracle*. URL: https://www.oracle.com/java/.

[30] U. Khedker, A. Sanyal, and B. Karkare. *Data Flow Analysis: Theory and Practice 1st*. 1st ed. CRC Press, Inc. Boca Raton, FL, USA, 2009, p. 395. ISBN: 0849328802, 9780849328800. URL: https://dl.acm.org/citation.cfm?id=1592955.

[31] G. A. Kildall. "A unified approach to global program optimization." In: *Proceedings of the 1st annual ACM SIGACT-SIGPLAN symposium on Principles of programming languages - POPL '73*. New York, New York, USA: ACM New York, NY, USA, 1973, pp. 194–206. DOI: 10.1145/512927.512945.

[32] G. E. Krasner and S. T. Pope. "A Cookbook for Using the Model- View-Controller User Interface Paradigm in Smalltalk-80." In: *Journal Of Object Oriented Programming (JOOP)* 1.3 (1988), pp. 26–49. URL: https://dl.acm.org/citation.cfm?id=50759.

[33] P. Krysiak, H. Lourenço, and J. Costa Seco. *Domain Specific Language for Data Validation in OutSystems Platform (MSc thesis)*. NOVA University of Lisbon, 2017.

[34] N. Lehmann and Tanter. "Gradual refinement types." In: *ACM SIGPLAN Notices* 52.1 (2017), pp. 775–788. ISSN: 03621340. DOI: 10.1145/3093333.3009856.

[35] H. Lourenço and R. Eugénio. "TrueChange™ under the hood: how we check the consistency of large models (almost) instantly." In: *Proceedings of the ACM/IEEE 22th International Conference on Model Driven Engineering Languages and Systems (MODELS)*. Munich, Germany, 2019.

[36] *Low-code Application Development Platform | Mendix*. URL: https://www.mendix.com/.

[37] S. C. McConnell. *Code Complete, Second Edition*. Vol. 136. 1. Microsoft Press, 2004, p. 3. ISBN: 0735619670, 9780735619678. URL: https://dl.acm.org/citation.cfm?id=1096143.

[38] B. Meyer. "Applying design by contract." In: *Computer* 25.10 (Oct. 1992), pp. 40–51. ISSN: 0018-9162. DOI: 10.1109/2.161279.

[39] B. Meyer. *Object-oriented Software Construction (2Nd Ed.)* 2nd ed. Prentice Hall PTR, 1997, p. 1254. ISBN: 0-13-629155-4. URL: https://dl.acm.org/citation.cfm?id=261119.

[40] *Microsoft SQL Server*. 2018. URL: https://www.microsoft.com/en-us/sql-server/sql-server-2017.

[41] *.NET | Free. Cross-platform. Open Source.* URL: https://dotnet.microsoft.com/.

[42] F. Nielson, H. R. Nielson, and C. Hankin. *Principles of Program Analysis*. Springer, Berlin, Heidelberg, 1999, p. 464. ISBN: 3540654100. URL: https://dl.acm.org/citation.cfm?id=555142.

[43] *Oracle Database*. URL: https://www.oracle.com/database/.

[44] *OutSystems tools and components*. URL: https://success.outsystems.com/Evaluation/Architecture/1_OutSystems_Platform_tools_and_components.

[45] F. Pfenning, A. Platzer, R. Simmons, and J. Hoffmann. *Lecture Notes on Liveness Analysis*. Carnegie Mellon University, United States, 2017. URL: https://www.cs.cmu.edu/~janh/courses/411/17/lec/05-liveness.pdf.

[46] B. C. Pierce. *Types and Programming Languages*. 1st. The MIT Press, 2002, p. 645. ISBN: 0262162091 9780262162098. URL: https://dl.acm.org/citation.cfm?id=509043.

[47] *Razor syntax reference for ASP.NET Core | Microsoft Docs*. URL: https://docs.microsoft.com/en-us/aspnet/core/mvc/views/razor?view=aspnetcore-2.2.

[48] *Ruby Programming Language*. URL: https://www.ruby-lang.org/en/.

[49] N. B. Ruparelia and N. B. "Software development lifecycle models." In: *ACM SIGSOFT Software Engineering Notes* 35.3 (May 2010), pp. 8–13. ISSN: 01635948. DOI: 10.1145/1764810.1764814.

[50] M. I. Schwartzbach and A. Møller. *Lecture Notes on Static Program Analysis*. University of Aarhus, Denmark, 2018. URL: https://cs.au.dk/~amoeller/spa/.

[51] R. W. Sebesta. *Concepts of Programming Languages*. 12th. Pearson, 2018. ISBN: 9780134997186.

[52] *Security Reports for OutSystems Factories (MSc thesis)*. NOVA University of Lisbon, 2019.

[53] *Service Studio Overview < Developing OutSystems Web Applications - Training - OutSystems*. URL: https://www.outsystems.com/learn/lesson/850/service-studio-overview/.

[54] H. S. S. Silberschatz Abraham; Korth F. *Database System Concepts, Sixth Edition*. 6th ed. 2010, p. 1349. ISBN: 978-0-07-352332-3. URL: https://www.db-book.com/db6/.

[55] M. Silva, H. Lourenço, and J. Costa Seco. "Preservação de invariantes de dados por geração automática de código de validação." In: *INForum 2019 Atas do 11º Simpósio de Informática*. 2019.

[56] Y. Smaragdakis and M. Bravenboer. "Using Datalog for Fast and Easy Program Analysis." In: *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*. 2011, pp. 245–251. ISBN: 9783642242052. DOI: 10/fzrgp8.

[57] *SWI-Prolog - Prolog language implementation*. URL: https://www.swi-prolog.org/.

[58] *Tabled execution (SLG resolution) - SWI-Prolog Documentation*. URL: https://www.swi-prolog.org/pldoc/man?section=tabling.

[59] *Validation Rules | Mendix Documentation*. URL: https://docs.mendix.com/refguide/validation-rules.

[60] N. Vazou, E. L. Seidel, R. Jhala, D. Vytiniotis, and S. Peyton-Jones. "Refinement types for Haskell." In: *ACM SIGPLAN Notices* 49.9 (Aug. 2014), pp. 269–282. ISSN: 03621340. DOI: 10.1145/2692915.2628161.

[61] N. Vazou, A. Bakst, and R. Jhala. "Bounded refinement types." In: *Proceedings of the 20th ACM SIGPLAN International Conference on Functional Programming - ICFP 2015*. New York, New York, USA: ACM Press, 2015, pp. 48–61. ISBN: 978-1-4503-3669-7. DOI: 10.1145/2784731.2784745.

[62] P. Vekris, B. Cosman, and R. Jhala. "Refinement types for TypeScript." In: *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation - PLDI 2016*. New York, New York, USA: ACM Press, 2016, pp. 310–325. ISBN: 9781450342612. DOI: 10.1145/2908080.2908110.

[63] E. Visser. "WebDSL: A Case Study in Domain-Specific Language Engineering." In: *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*. Springer, 2008, pp. 291–373. ISBN: 978-3-540-88643-3. DOI: 10/b8r9gx.

[64] *WebDSL - Domain-Specific Language for Web Applications*. URL: https://webdsl.org/.

[65] M. Young and M. Pezze. *Software Testing and Analysis: Process, Principles and Techniques*. 1st. John Wiley & Sons, Inc. , USA, 2005, p. 542. ISBN: 978-0-471-45593-6. URL: https://dl.acm.org/citation.cfm?id=1076948.

Appendix A

Facts Generated

```
rule(Room, Rule1: AdultsCapacity > 0).
rule(Room, Rule2: ChildrenCapacity >= 0).
rule(Room, Rule3: Price > 0).
rule(Room, Rule4: RoomNumber > 0).

follows(webPage, start).
follows(start, conditionalStructure1).
follows(assignNode1, conditionalStructure2).
follows(assignNode2, createRecord).
follows(createRecord, end).
followsIf(conditionalStructure1, assignNode1, (RoomForm.Room.AdultsCapacity > 10)).
followsIf(conditionalStructure1, conditionalStructure2, not(RoomForm.Room.AdultsCapacity
    ↪ > 10)).
followsIf(conditionalStructure2, assignNode2, (RoomForm.Room.ChildrenCapacity > 5)).
followsIf(conditionalStructure2, createRecord, not(RoomForm.Room.ChildrenCapacity > 5)).

assign(assignNode1, Room.AdultsCapacity, 10).
assign(assignNode2, Room.ChildrenCapacity, 5).

input(createRecord, Room, RoomNumber, RoomForm.Room.RoomNumber).
input(createRecord, Room, AdultsCapacity,RoomForm.Room.AdultsCapacity).
input(createRecord, Room, ChildrenCapacity, RoomForm.Room.ChildrenCapacity).
input(createRecord, Room, Price, RoomForm.Room.Price).

node(webpage).
node(start).
node(conditionalStructure1).
node(conditionalStructure2).
node(end).
```

Listing A.1: Example *Rule* facts.

Appendix B

OutSystems Meta-model

This meta-model describes a simplification of the actual OutSystems meta-model. The meta model's meta model is presented in [35].

```
<MetaModel xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:xsd="http://www.w3.org/2001/XMLSchema"
    xsi:schemaLocation="http://www.outsystems.comMetaModel.xsd">

    <Class name="ESpace">
        <Property name="Name" type="Text" />
        <Child name="WebScreens" type="Screen" />
        <Child name="Actions" type="Action" />
        <Child name="Entities" type="Entity" />
    </Class>

    <Class name="Screen">
        <Property name="Name" type="Text" />
        <Child name="InputParameters" type="InputParameter" />
        <Child name="LocalVariables" type="LocalVariable" />
        <Child name="Widgets" type="Widget" />
        <Child name="ScreenActions" type="Action" />
    </Class>

    <Class name="Action">
        <Property name="Name" type="Text" />
        <Child name="InputParameters" type="InputParameter" />
        <Child name="LocalVariables" type="LocalVariable" />
        <Child name="Nodes" type="ActionNode" />
    </Class>

    <Class name="ActionNode">
        <Property name="Target" type ="ActionNode" />
    </Class>
```

```
30
31      <Class name="Start" base="ActionNode" />
32      <Class name="End" base="ActionNode" />
33
34      <Class name="Assign" base="ActionNode">
35          <Child name="Assignments" type="Assignment" />
36      </Class>
37
38      <Class name="If">
39          <Property name="TrueLink" type ="ActionNode" />
40          <Property name="FalseLink" type ="ActionNode" />
41          <Property name="Condition" type="Expression" />
42      </Class>
43
44      <Class name="Execute" base="ActionNode">
45          <Property name="Action" type="Action" />
46          <Child name="Arguments" type="Argument" />
47      </Class>
48
49      <Class name="Argument" verifyDependencies="Parameter.IsMandatory, Parameter.Type">
50          <Property name="Parameter" type="InputParameter" />
51          <Property name="Value" type="Expression" isOptional="true" />
52      </Class>
53
54      <Class name="Entity">
55          <Property name="Name" type="Text" />
56          <Child name="Attributes" type="Attribute" />
57          <Child name="EntityActions" type="EntityAction" />
58      </Class>
59
60      <Class name="Attribute">
61          <Property name="Name" type="Text" />
62          <Property name="DataType" type="Type" />
63      </Class>
64  </MetaModel>
```

Listing B.1: Simplification of the OutSystems meta-model.

Appendix C

Solutions Obtained

```
1  // * = hasRuleOut (for simplicity)
2
3  *(webPage, (RoomForm.Room.AdultsCapacity > 10 and RoomForm.Room.ChildrenCapacity > 5)
       ↪ ⟹ Rule1: 10 > 0)
4  *(webPage, (RoomForm.Room.AdultsCapacity > 10 and RoomForm.Room.ChildrenCapacity > 5)
       ↪ ⟹ Rule2: 5 >= 0)
5  *(webPage, (RoomForm.Room.AdultsCapacity > 10 and RoomForm.Room.ChildrenCapacity > 5)
       ↪ ⟹ Rule3: RoomForm.Room.Price > 0)
6  *(webPage, (RoomForm.Room.AdultsCapacity > 10 and RoomForm.Room.ChildrenCapacity > 5)
       ↪ ⟹ Rule4: RoomForm.Room.RoomNumber > 0)
7
8  *(webPage, (RoomForm.Room.AdultsCapacity > 10 and not(RoomForm.Room.ChildrenCapacity > 5)
       ↪ ) ⟹ Rule1: 10 > 0)
9  *(webPage, (RoomForm.Room.AdultsCapacity > 10 and not(RoomForm.Room.ChildrenCapacity > 5)
       ↪ ) ⟹ Rule2: RoomForm.Room.ChildrenCapacity >= 0)
10 *(webPage, (RoomForm.Room.AdultsCapacity > 10 and not(RoomForm.Room.ChildrenCapacity > 5)
       ↪ ) ⟹ Rule3: RoomForm.Room.Price > 0)
11 *(webPage, (RoomForm.Room.AdultsCapacity > 10 and not(RoomForm.Room.ChildrenCapacity > 5)
       ↪ ) ⟹ Rule4: RoomForm.Room.RoomNumber > 0)
12
13 *(webPage, (not(RoomForm.Room.AdultsCapacity > 10) and RoomForm.Room.ChildrenCapacity >
       ↪ 5) ⟹ Rule1: RoomForm.Room.AdultsCapacity > 0)
14 *(webPage, (not(RoomForm.Room.AdultsCapacity > 10) and RoomForm.Room.ChildrenCapacity >
       ↪ 5) ⟹ Rule2: 5 >= 0)
15 *(webPage, (not(RoomForm.Room.AdultsCapacity > 10) and RoomForm.Room.ChildrenCapacity >
       ↪ 5) ⟹ Rule3: RoomForm.Room.Price > 0)
16 *(webPage, (not(RoomForm.Room.AdultsCapacity > 10) and RoomForm.Room.ChildrenCapacity >
       ↪ 5) ⟹ Rule4: RoomForm.Room.RoomNumber > 0)
17
18 *(webPage, (not(RoomForm.Record.Room.AdultsCapacity > 10) and not(RoomForm.Record.Room.
       ↪ ChildrenCapacity > 5)) ⟹ Rule1: RoomForm.Room.AdultsCapacity > 0)
```

```
19 *(webPage, (not(RoomForm.Record.Room.AdultsCapacity > 10) and not(RoomForm.Record.Room.
       ↪ ChildrenCapacity > 5)) ⟹ Rule2: RoomForm.Room.ChildrenCapacity > 5)
20 *(webPage, (not(RoomForm.Record.Room.AdultsCapacity > 10) and not(RoomForm.Record.Room.
       ↪ ChildrenCapacity > 5)) ⟹ Rule3: RoomForm.Room.Price > 0)
21 *(webPage, (not(RoomForm.Record.Room.AdultsCapacity > 10) and not(RoomForm.Record.Room.
       ↪ ChildrenCapacity > 5)) ⟹ Rule4: RoomForm.Room.RoomNumber > 0)
```

Listing C.1: Solutions obtained from the static analysis.